PART 2

Question 1: In the worse case there is one even more radical debugging technique used to isolate a problem, what is it? Explain.

Ans : The radical solution is to identify and isolate areas of problems by the following steps

- Knowing where the bug occurs, isolate files or scripts responsible for it. The exact point when it happens may even help in zeroing in on the method responsible for it.
- This method might be a bundle of three or more parts/methods
- Comment out each portion to see when the bug is removed.
- Applying this approach recursively on methods or functions should help you identify why the error/bug is occuring.

Question 2: As a project grows larger and the code base more extensive, the amount of bugs and their severity grows. Explain what causes this phenomenon.

Coupling. It defines to what extent a piece of code is dependent on another. Higher amount of coupling between different areas of the games results in trickling of bugs to other components and modules.

Question 3: Can you think of any solution(s) to the above problem? Explain how you would execute this.

Less coupling, more modular code. Each component should have a defined contract(interface and methods used for
interaction with it) and hence it should behave like a black box. This design is usually called Separation of concern

PART 3

A1

Question 1: Explain the merits of a pure function

Answer 1 : Pure functions are something that work like an honest interface - They return values based only on the arguments provided and don't modify any references. Their results never change for same arguments under different times or any other random factors.

Hence they provide more cohesion or least coupling since their results are independent of other modules.

The advantage of a pure function is that they can be coupled or pipelined with other pure function in a predictable way.

Ex : sum(a,b) and mult(a,b) are pure functions and hence any pure combination of these two will also be a pure function.

Question 2: Explain what this function(Validsequence) does?

Answer 2: It validates a sequence greater than 2 and returns the score earned by merging **(same or WILD)** number tiles using shift bit operations.

Since each tile can be merged with a tile equal to its own value or wild, we just use powers of 2 to calculate score

Question 3: Add in missing comments for lines on this function without comments.
Answer 3 :Done on Editor

A2

Question 1: Read, understand, and add comments to the function isValidTile().
Answer 1 : Done on Editor

Question 2: It is generally discouraged to use goto statements in modern code, explain why.

Answer 2 : They make the code harder to read and undermine the concept of structured programming which defines how the control of flow is transferred using exception handling, decision making and iterations

Question 3: Does the common wisdom about goto statements apply here? Explain

Answer 3 **:** Yes, if by common wisdom we mean that goto statements can be replaced by a set of loops, if else blocks and a combination of structured constructs

BEFORE

```
bool Match2TestWnd::isValidTile(int i, int j, gameState * state)
{
	state->sequence.push_back(state->grid[i][j]);

	if (state->sequence.size() == 1) goto TILE_VALID;

	if (state->selectOrdr[i][j] != NOT_SELECTED) goto TILE_INVALID;

	if (validateSequence(&state->sequence) == 0) goto TILE_INVALID;

	if ((i > 0) && (state->sequence.size() - 2 == state->selectOrdr[i - 1][j] && state->selectOrdr[i - 1][j] != NOT_SELECTED)) goto TILE_VALID;
	if ((i < 4) && (state->sequence.size() - 2 == state->selectOrdr[i + 1][j] && state->selectOrdr[i + 1][j] != NOT_SELECTED)) goto TILE_VALID;
	if ((j > 0) && (state->sequence.size() - 2 == state->selectOrdr[i][j - 1] && state->selectOrdr[i][j - 1] != NOT_SELECTED)) goto TILE_VALID;
	if ((j < 4) && (state->sequence.size() - 2 == state->selectOrdr[i][j + 1] && state->selectOrdr[i][j + 1] != NOT_SELECTED)) goto TILE_VALID;

	TILE_INVALID:
	state->sequence.pop_back();
	return false;

	TILE_VALID:
	state->sequence.pop_back();
	return true;

}
```

AFTER

```
bool Match2TestWnd::isValidTile(int i, int j, gameState * state)
{
```

```
		state->sequence.push_back(state->grid[i][j]);

		if (state->sequence.size() == 1
			|| (i > 0) && (state->sequence.size() - 2 == state->selectOrdr[i - 1][j] &&
state->selectOrdr[i - 1][j] != NOT_SELECTED)
			|| (i < 4) && (state->sequence.size() - 2 == state->selectOrdr[i + 1][j] &&
state->selectOrdr[i + 1][j] != NOT_SELECTED)
			|| (j > 0) && (state->sequence.size() - 2 == state->selectOrdr[i][j - 1] &&
state->selectOrdr[i][j - 1] != NOT_SELECTED)
			|| (j < 4) && (state->sequence.size() - 2 == state->selectOrdr[i][j + 1] &&
state->selectOrdr[i][j + 1] != NOT_SELECTED)
			)
		{
			state->sequence.pop_back();
			return true;
		}
		else if ((state->selectOrdr[i][j] != NOT_SELECTED)
			|| (validateSequence(&state->sequence) == 0)
			)
		{
			state->sequence.pop_back();
			return false;
		}


}
```