My program starts with the main procedure by taking input from user as in the example below. It checks if the given numbers are greater than 0, and the given characters are only '.' or 'O'.

```
Please enter row number (row > 0): 3
Please enter col number (col > 0): 4
Please enter number of times (n > 0): 3
.O..
.OO.
...O
```

After row, number and n inputs are read, the game board and time board are created as 1D array dynamically. While taking characters for the initial game board, the time board is also filled with either 0s or -1s. -1 means there is no bomb, other numbers indicate the time passed after a bomb is planted.

After all inputs are taken, "playGame" procedure is called. It's a procedure to play the game till it reaches the n. It takes row, col, n, game board address and time board address as parameters from register $a0-$a3 and $t9 consequently. It stores the return address and the s registers that it will use into the stack, then it puts back them at the end of the procedure. The procedure first calls "incrementBombTimes" procedure to increment the initial bombs' times and then starts the loop by the time starting with 2 going to n. Inside the loop, it calls "incrementBombTimes" and checks whether the time can be divided by 2. If it is the case, it calls "putBombsToEmptyGrids" procedure, otherwise it calls "detonateBombs" procedure. ("playGame" C version is on the next page.)

The "detonateBombs" procedure takes row, col, game board address and time board address as parameters. Firstly, it iterates through the boards and checks whether there is a bomb whose time is exactly 3 by looking at the time board. If it is, then it detonates this bomb and its four surrounding grids while controlling the board lines. After this loop finishes, it's time to clear the time board. There is one more iteration which iterates through the game and time board and clears the time board by checking whether there is a bomb in the game board. If there is no bomb in the game board for the current position, then time board's value is assigned to -1. Then it returns to the caller. There is no return value since the changes are made directly in the memory. ("detonateBombs" C version is on the next page.)

The "putBombsToEmptyGrids" procedure takes row, col, game board address and time board address as parameters. It iterates through the game and time board, each time checks whether there is no bomb for the current grid by checking if the time value is -1. If it is the case, then it puts '.' to game board, and 0 to the time board. Then it returns to the caller. There is no return value since the changes are made directly in the memory. ("putBombsToEmptyGrids" C version is on the next page.)

The " incrementBombTimes" procedure takes row, col and time board address. It iterates through the time board, every time it checks if there is a bomb in the current grid. If it's the case, it increments this grid's time value. Then it returns to the caller. There is no return value since the changes are made directly in the memory. ("incrementBombTimes" C version is on the next page.)

After the game is done, "playGame" procedure jumped to the caller procedure "main". The main procedure then calls "printMatrix" procedure to print the game board. Finally, it exits the program.

The "printMatrix" procedure takes row, col and game board address. It first stores the s registers that it will use into the stack, at the end it restores these values. It iterates through the given game board and prints the characters in the board. Then it returns to the caller. ("printMatrix" C version is on next page.)

```
void playGame(char* gameBoard, int* timeBoard, int row, int col, int n) {
    // 1st second. n  = 1
    incrementBombTimes(timeBoard, row, col);

    int i;
    for (i = 2; i <= n; ++i) {
        // Increment bombs all time.
        incrementBombTimes(timeBoard, row, col);

        if (i % 2 == 0) {
            putBombsToEmptyGrids(gameBoard, timeBoard, row, col);
        }
        else {
            detonateBombs(gameBoard, timeBoard, row, col);
        }
    }
}
```

```
void detonateBombs(char* gameBoard, int* timeBoard, int row, int col) {
    // Detonates the bombs whose time is exactly 3.
    int i;
    for (i = 0; i < row; ++i) {
        int j;
        for (j = 0; j < col; ++j) {
            if (timeBoard[i * col + j] == 3) {
                // Detonate the bomb itself.
                gameBoard[i * col + j] = '.';

                if (i > 0) {
                    gameBoard[(i - 1) * col + j] = '.';
                }
                if (i < row - 1) {
                    gameBoard[(i + 1) * col + j] = '.';
                }
                if (j > 0) {
                    gameBoard[i * col + (j - 1)] = '.';
                }
                if (j < col - 1) {
                    gameBoard[i * col + (j + 1)] = '.';
                }
            }
        }
    }

    for (i = 0; i < row; ++i) {
        int j;
        for (j = 0; j < col; ++j) {
            if (gameBoard[i * col + j] == '.') {
                timeBoard[i * col + j] = -1;
            }
        }
    }
}
```

```c
void putBombsToEmptyGrids(char* gameBoard, int* timeBoard, int row, int col) {
    int i;
    for (i = 0; i < row; ++i) {
        int j;
        for (j = 0; j < col; ++j) {
            if (timeBoard[i * col + j] == -1) {
                gameBoard[i * col + j] = 'O';
                timeBoard[i * col + j] = 0;
            }
        }
    }
}

void incrementBombTimes(int* timeBoard, int row, int col) {
    int i;
    for (i = 0; i < row; ++i) {
        int j;
        for (j = 0; j < col; ++j) {
            if (timeBoard[i * col + j] != -1) {
                ++timeBoard[i * col + j];
            }
        }
    }
}


void printGameBoard(char* gameBoard, int row, int col) {
    // Print read matrix
    int i = 0;
    for (i = 0; i < row; ++i) {
        int j = 0;
        for (j = 0; j < col; ++j) {
            printf("%c", gameBoard[i * col + j]);
        }
        printf("\n");
    }
}
```