Mastermind Recursion
Project 2

Bryan Estrada

CSC-17C

Fall 2024

Project 2

12/08/2024

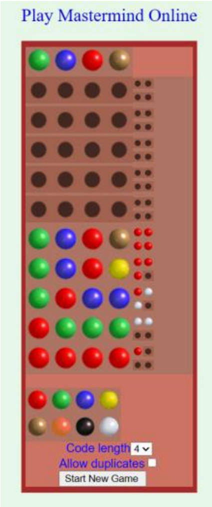
Table of Contents

Introduction

Mordecai Meirowitz, an Israeli postmaster, and telecommunications expert created the game Mastermind in 1970. Mastermind is a two-player code-breaking game. It bears similarities to a traditional pencil and paper game called Bulls and Cows, which might have existed for over a century. Mastermind is a game of deduction and logic, and it has been popular for decades due to its simple yet challenging gameplay.

How the Online Game Version Works

The game is played between two players: one player is an AI that creates a secret code, and the other player is a human user that tries to guess the code within a certain number of turns. The game consists of a decoding board and code pegs of different colors. The codemaker selects a secret code and places the pegs in a specific order on the board, hidden from the user. The user then makes a series of guesses, and after each guess, the AI provides feedback in the form of colored pegs. A red peg indicates a correct color in the correct position, and a white peg indicates a correct color in the wrong position. The game continues until the user guesses the correct code or until a predetermined number of turns have been exhausted.



Play Mastermind Online

Code length 4

Allow duplicates □

My Approach to the Game

Similarities to the Board Game

My version of the game and the actual game are similar in a few ways. Both games are played by a user and an AI. The AI generates a random code according to user's settings (duplicates or not, code length). The user input their guess and the AI provides hints each turn. Both games consists of 10 turns. Lastly, both games offers an option for the user to read the rules of the games and how the game gives the user hints.

Differences from the Board Game

The games differ in the pegs. The online version consists of a code of color pegs, and the user has to guess the correct sequence of color pegs the AI generates. On the other hand, my program consists of numbers, the AI generates a sequence of digits from 1 – 8, and the user has to input their guess using numbers. Additionally, the game offers statistics of the game like the amount of victories and losses depending on the code length (4, 6, or 8) and if the user played with duplicates or not.

GitHub Repository Link

https://github.com/Bryan-EstradaC/CSC17C_Project_2

Number of Lines

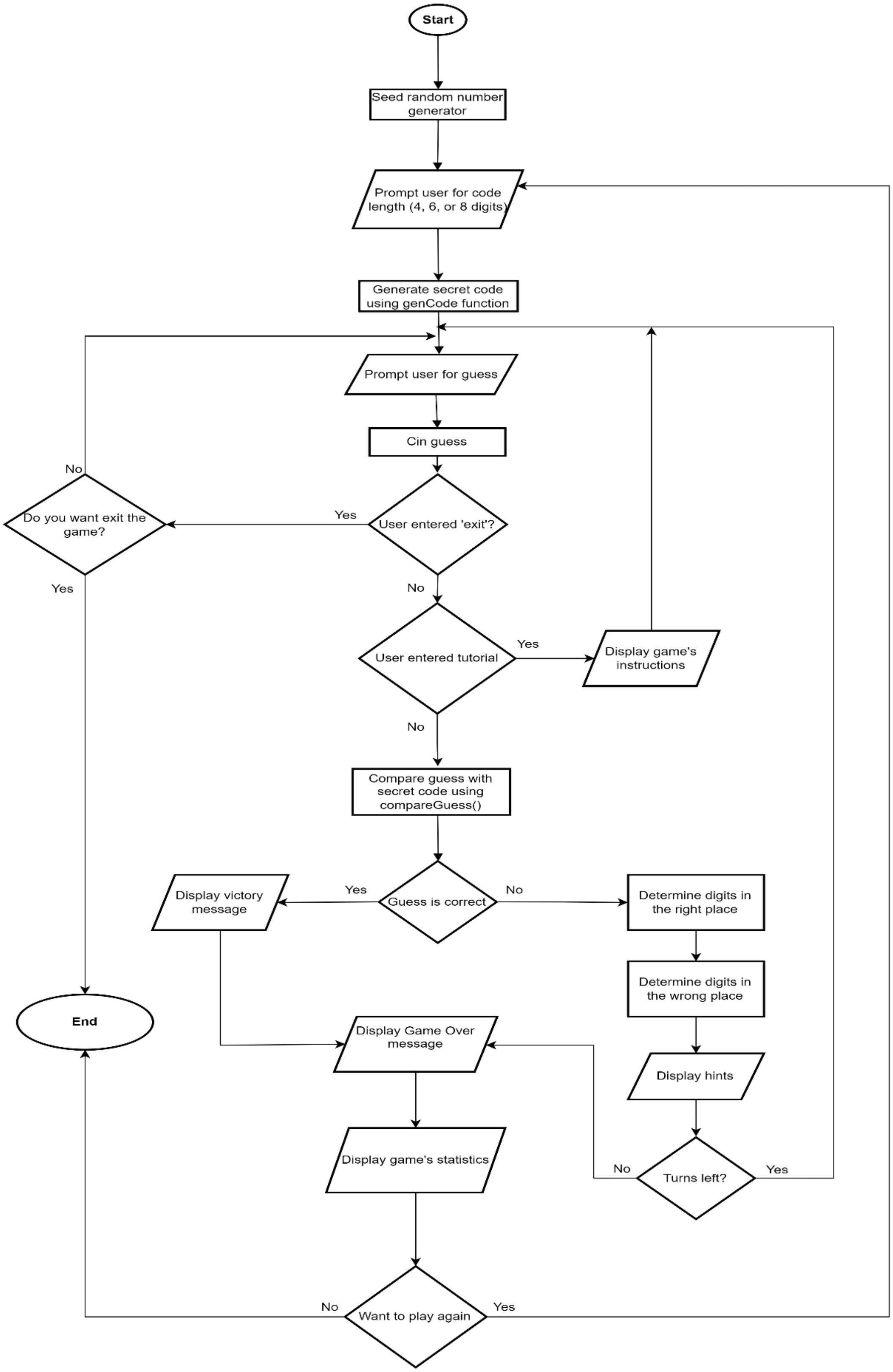
757 lines

The Logic of it All

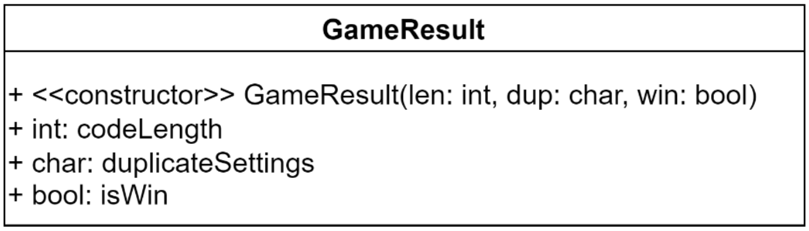
Pseudocode

```
Start Game

Call setupGame()
Print welcome message

Repeat while playAgain is 'y' and quit is false:
  Set endGame to false
  Set skipTurn to false
  Convert playAgain to lowercase

  If playAgain is 'y':
    Initialize turns with values from 1 to 10
    Get valid code length and store in length
    Get valid choice for duplicates and store in choiceDuplicate
    Generate code and store in code
    Print the generated code

    Print instructions for the user

    While the game is not ended, turns are available, and quit is false:
      Reset skipTurn to false at the start of each turn
      Print options to exit or show tutorial
      Prompt the user to input their guess

      If the input is "exit":
        Call exitingGame to ask user for confirmation
        Set skipTurn to true to skip the rest of the loop

      If the input is "tutorial":
        Show game instructions
        Set skipTurn to true

      If skipTurn is false:
        Validate the guess input

      If skipTurn is false:
        Compare the guess with the code

    If the game is not quit:
      Show game over message
      Display statistics

    Clear the code list

While playAgain is 'y' and quit is false

End Game
```

Flowchart

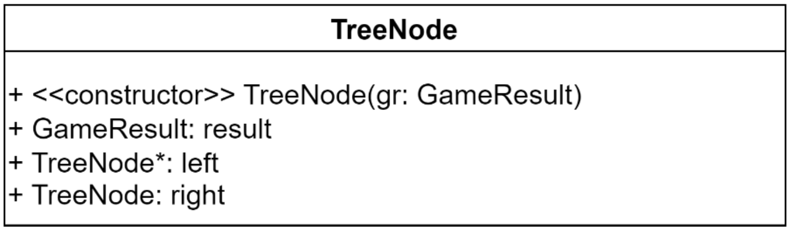

Start
Seed random number generator
Prompt user for code length (4, 6, or 8 digits)
Generate secret code using genCode function
Prompt user for guess
Cin guess
User entered 'exit'?
Yes
Do you want exit the game?
No
Yes
No
User entered tutorial
Yes
Display game's instructions
No
Compare guess with secret code using compareGuess()
Guess is correct
Yes
Display victory message
No
Determine digits in the right place
Determine digits in the wrong place
Display hints
Turns left?
No
Yes
End
Display Game Over message
Display game's statistics
Want to play again
No
Yes

UML Diagrams



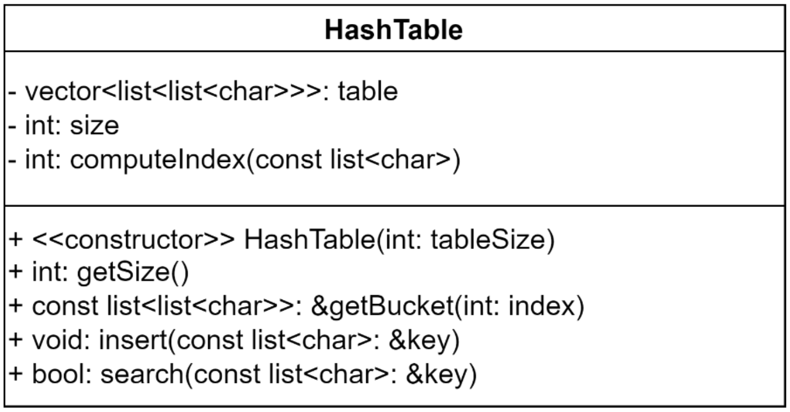
GameResult

```
+ <<constructor>> GameResult(len: int, dup: char, win: bool)
+ int: codeLength
+ char: duplicateSettings
+ bool: isWin
```



TreeNode

```
+ <<constructor>> TreeNode(gr: GameResult)
+ GameResult: result
+ TreeNode*: left
+ TreeNode: right
```



HashTable

```
- vector<list<list<char>>>: table
- int: size
- int: computeIndex(const list<char>)

+ <<constructor>> HashTable(int: tableSize)
+ int: getSize()
+ const list<list<char>>: &getBucket(int: index)
+ void: insert(const list<char>: &key)
+ bool: search(const list<char>: &key)
```

List of Main Variables

STL:
Containers:

- **list<char> code:** A container that holds a string of chars that represents the secret code generated by the computer.
- **list<char> guess:** A container that holds a string of chars representing the guess entered by the user.
- **stack<int> turns:** Container that holds the number of turns that the user has left each time he or she enters their guess. Since it is a stack, it is initialized from 1 to 10 (turns). Then, 10 will be the element on the top of the stack, representing that the user will have 10 turns at the beginning of the game. When the user used one of their turns by entering a valid guessing, the program will pop the top turn from the stack until the user runs out of turns (turns = 0).

- **set<char> unique_number:** Container that holds unique numbers to help generate the code without duplicates. Since sets doesn’t accept repeating elements, unique_number will hold numbers obtained from another STL container that is being shuffled until the code reaches the selected size by the user.
- **deque<char> numbers:** Container that holds the number from 1 to 8, and it is shuffled to generate a random code with duplicates or without duplicates.
- **map<char, int> code_count:** Container that holds elements from the user’s guess that list<char> code also contains but user entered in the wrong position. code_count will contain the element as a key char, and the numbers of times the element appeared in the user’s guess. This will help to count how many times an element was in the wrong from the user's guess even when the user’s is playing with duplicates elements. Since maps only accepts unique elements, then code_count will hold the times a number was entered in the wrong position. code_count will help to give hints to the user on what possible numbers are entered in the correct or wrong position.
- **queue<GameResult> resultsQueue:** Container that holds a struct with the results of a game (length of the code, duplicates or not duplicates, and game win or lose). The program uses this container to store and print statistics of the user’s games. For instance, it will help to store and print how many victories and losses the user has when he or she has played with a code length of 4 with duplicates. Since queues operates in a FIFO type of arrangement, once the first element is stored and displayed, the element is popped from the back, leaving space for the next struct with the results of the next game.

Iterators:

- **Forward iterator:** The program uses forward iterator when tries to access elements from the lists. In the function hint(), code_it and guess_it are initialized with the first element of the list containers code and guess, respectively. Since the function needs to compare each element of the containers, once the comparison is done, the iterator is incremented by one using the increment operator (++).

Algorithms:

- **find:** The program uses find() twice. find is used in the function genCode() to find an element in the set unique_numbers and determine if the element is not in the container, if it is not in the container, so the element can be stored in the list code when the user chose not duplicates. The second time find() is used, it is in the function hint() to find an element from the list guess, once the element is found, the function determines if the found element is in the list code.
- **Random_Shuffle:** It shuffles the elements of the container numbers in the function genCode() every time a while-loop runs, this will guarantee a better randomization of the elements stored in the list code, especially when the user chooses to play with duplicates.
- **max_element and min_element:** It determines what type of game the user has obtained more victories and fewer victories from (Duplicates or No-Duplicates) in the function displayStatistics().

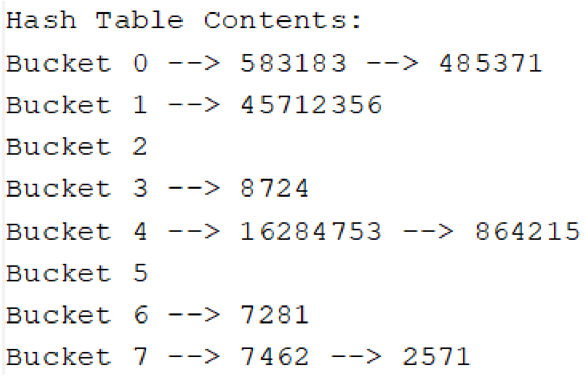
Updates since Project 1

Recursive Functions

- **genNums():** This function generates random numbers from 1 to 8 to store them in the code, which it does it recursively. Also, it checks if the user selected to play with or without duplicates to determine if numbers generated should be repeated.
- **hint_recursive():** This functions scans recursively the elements from the user's guess to determine if the element is in the same position as it should be in the code.
- **hint_recursive_misplaced():** This function scans the elements from the user's guess recursively to determine if the element is in the code, too, but in the wrong position. Also, it prints hints for the user based on the number of elements in the correct and incorrect position on the user's guess.

Hashing

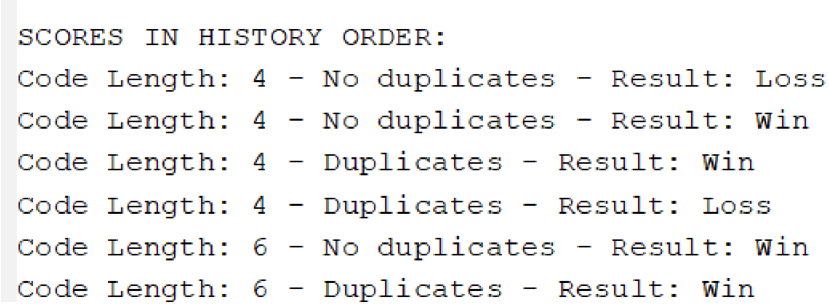
Every time the program generates a code the user has to guess; the code is stored in a hashing table. The hashing table receives the code and passes it to RSHash(), which is a hash function, and when it is returned, it is modded according to the hash table size. This hash table will be helpful to have a codes-generated history of the game, and also will be helpful to search for codes with order O(1). The hash table utilizes chaining in order to handle collisions. See example below where codes of different length has been stored after nine games:



```
Hash Table Contents:
Bucket 0 --> 583183 --> 485371
Bucket 1 --> 45712356
Bucket 2
Bucket 3 --> 8724
Bucket 4 --> 16284753 --> 864215
Bucket 5
Bucket 6 --> 7281
Bucket 7 --> 7462 --> 2571
```

Binary Tree

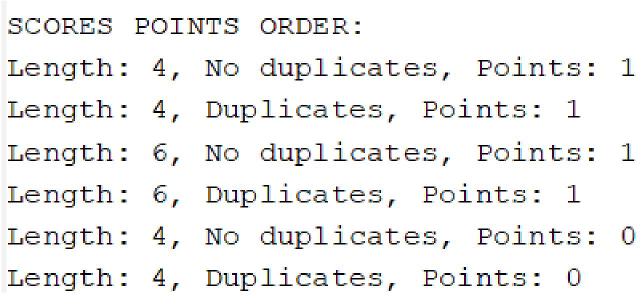
The program, now, stores the number of victories and losses in a binary tree, which then displays statistics of the user traversing the tree in order according to the code length.



```
SCORES IN HISTORY ORDER:
Code Length: 4 - No duplicates - Result: Loss
Code Length: 4 - No duplicates - Result: Win
Code Length: 4 - Duplicates - Result: Win
Code Length: 4 - Duplicates - Result: Loss
Code Length: 6 - No duplicates - Result: Win
Code Length: 6 - Duplicates - Result: Win
```

Recursive Sort

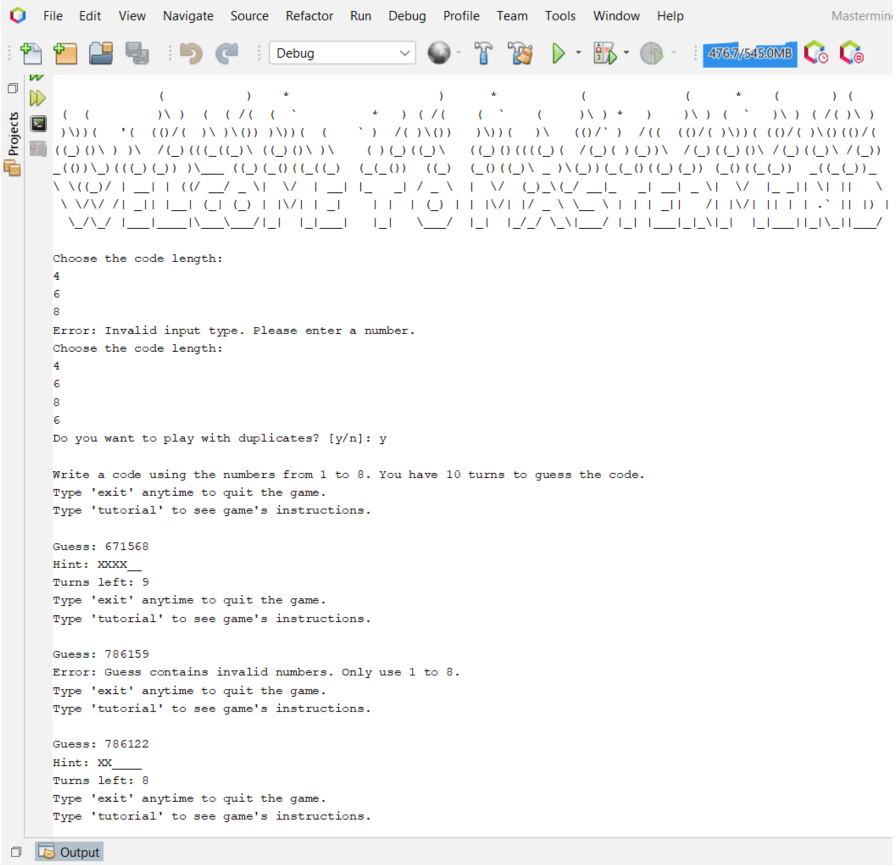
- **Merge Sort:** Alternatively, the program also prints statistics of the game using merge sort which arrange the results of the games from wins to losses. In other words, it shows first the wins, and then the losses (1 win = 1 point).



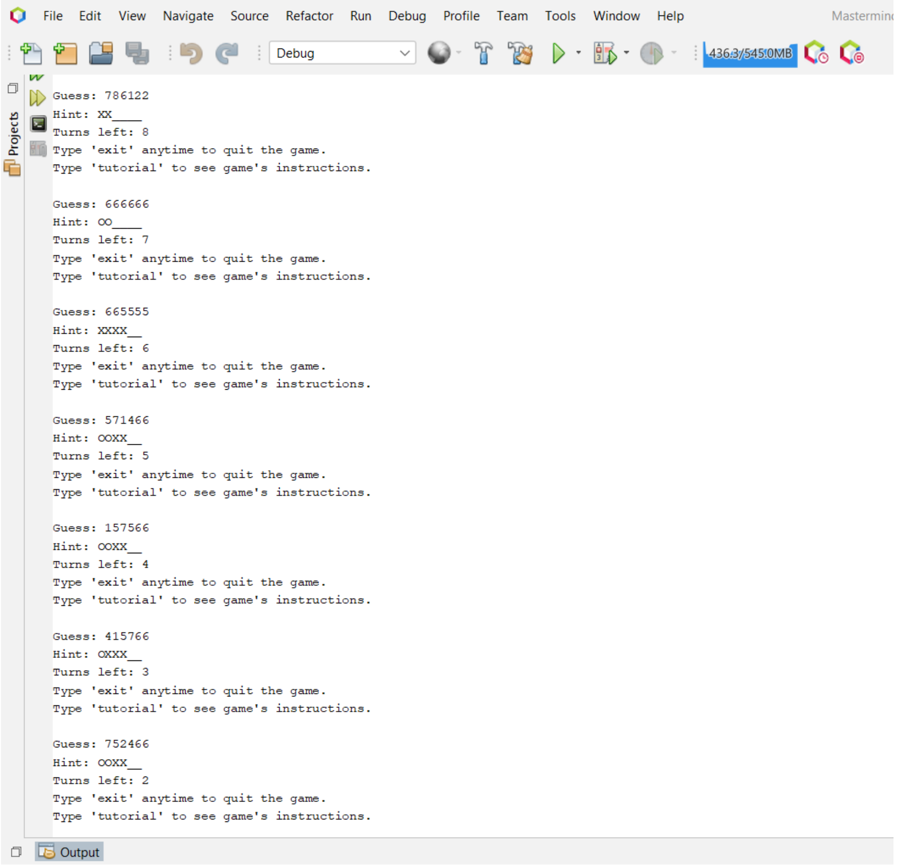
```
SCORES POINTS ORDER:
Length: 4, No duplicates, Points: 1
Length: 4, Duplicates, Points: 1
Length: 6, No duplicates, Points: 1
Length: 6, Duplicates, Points: 1
Length: 4, No duplicates, Points: 0
Length: 4, Duplicates, Points: 0
```

Proof of a Working Product

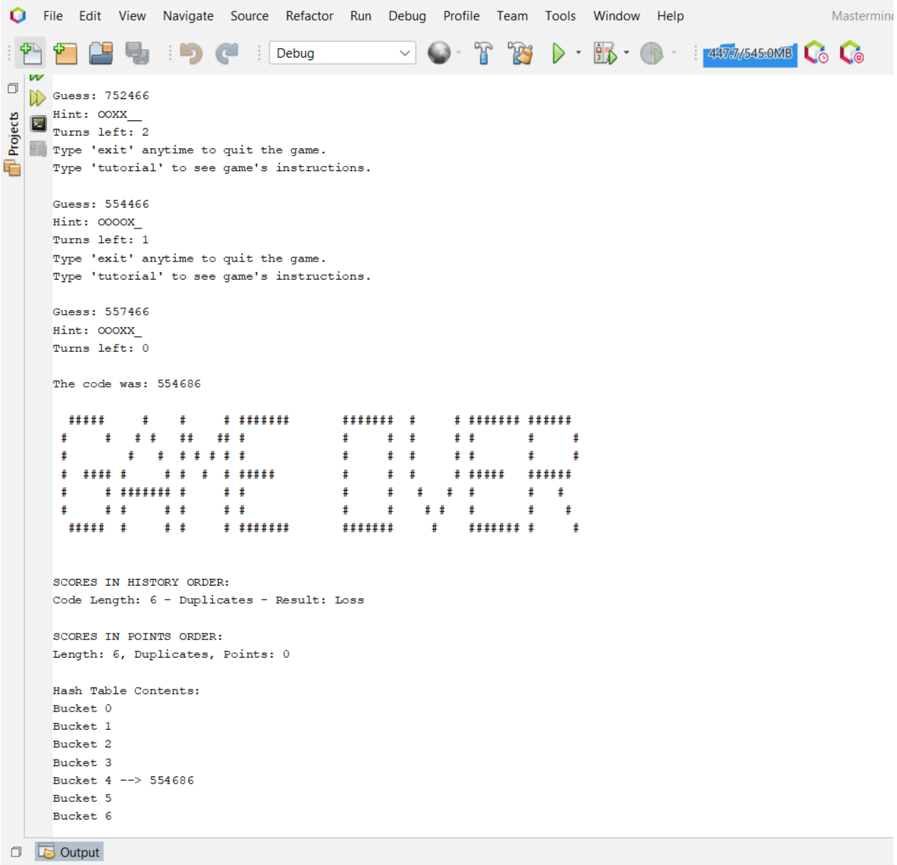
This is an example of the program executing, the program gives accurate feedback.



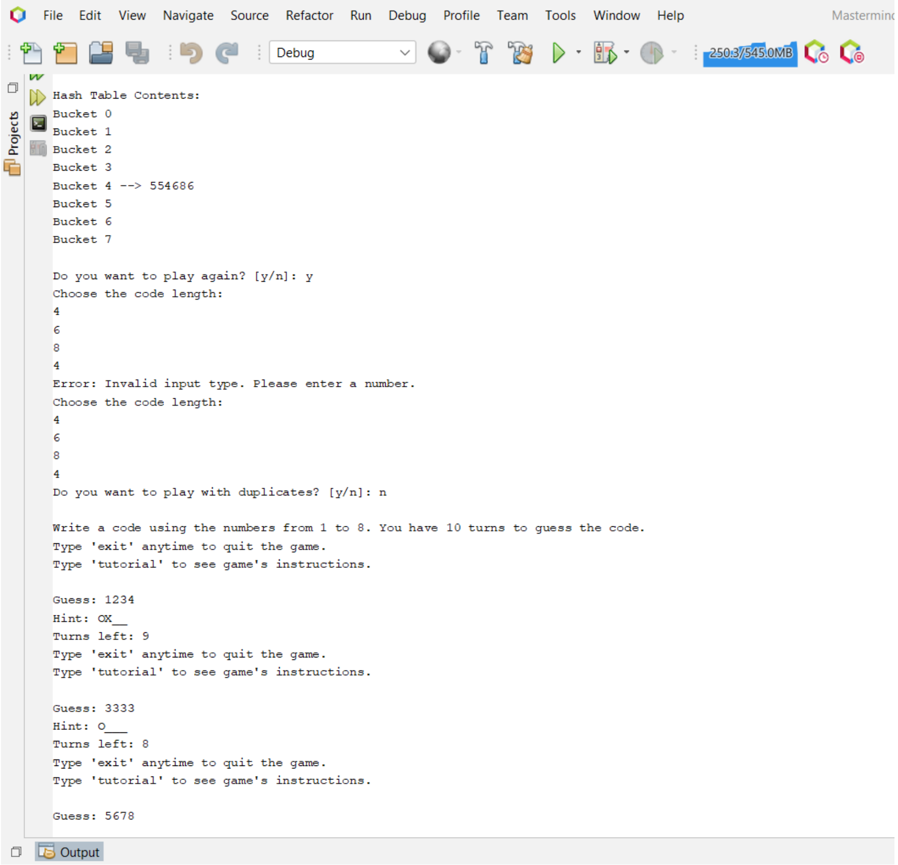
```
               (              )     *                       )         *                (                 (        *      (          )  (
 (    (             )\ )    (    ( /(    (   `              *    )  ( /(       (   `       (        )\ )  *    )        )\ )   (   `     )\ )   ( /(  )\ )
 )\))(     '(    (()/(    )\ )\())  )\))(    (       ` )    /(  )\())      )\))(     )\      (()/` )    /((    (()/(  )\))(  (()/(   )\()  (()/(
((_)()\ )  )\   /(_)((((_((_)\  ((_)()\ )\       ( )(_)((_)\      ((_)()((((_)(   /(_)( )(_))\   /(_)((_)()\ /(_)((_)\ /(_))
_(())\_)(((_)(_)) )\___ ((_)(_()((_((_)   (_(_())   ((_)     (_()((_)\ _ )\(_))(_(_()((_)(_))   (_()((_(_))   _((_(_))_
\ \((_)/ | __| | ((/ __/ _ \|  \/  | __| |_   _| / _ \    |  \/  |(_)_\(_/ __|_   _| __| _ \|  \/  |_ _|| \| ||   \
 \ \/\/ /| _|| |__| (_| (_) | |\/| | _|    | |  | (_) |   | |\/| | / _ \ \__ \ | | | _||   /| |\/| || | | .` || |) |
  \_/\_/ |___|____|\___\___/|_|  |_|___|   |_|   \___/    |_|  |_|/_/ \_\|___/ |_| |___|_|_\|_|  |_|___||_|\_||___/

Choose the code length:
4
6
8
Error: Invalid input type. Please enter a number.
Choose the code length:
4
6
8
6
Do you want to play with duplicates? [y/n]: y

Write a code using the numbers from 1 to 8. You have 10 turns to guess the code.
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 671568
Hint: XXXX__
Turns left: 9
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 786159
Error: Guess contains invalid numbers. Only use 1 to 8.
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 786122
Hint: XX____
Turns left: 8
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
```



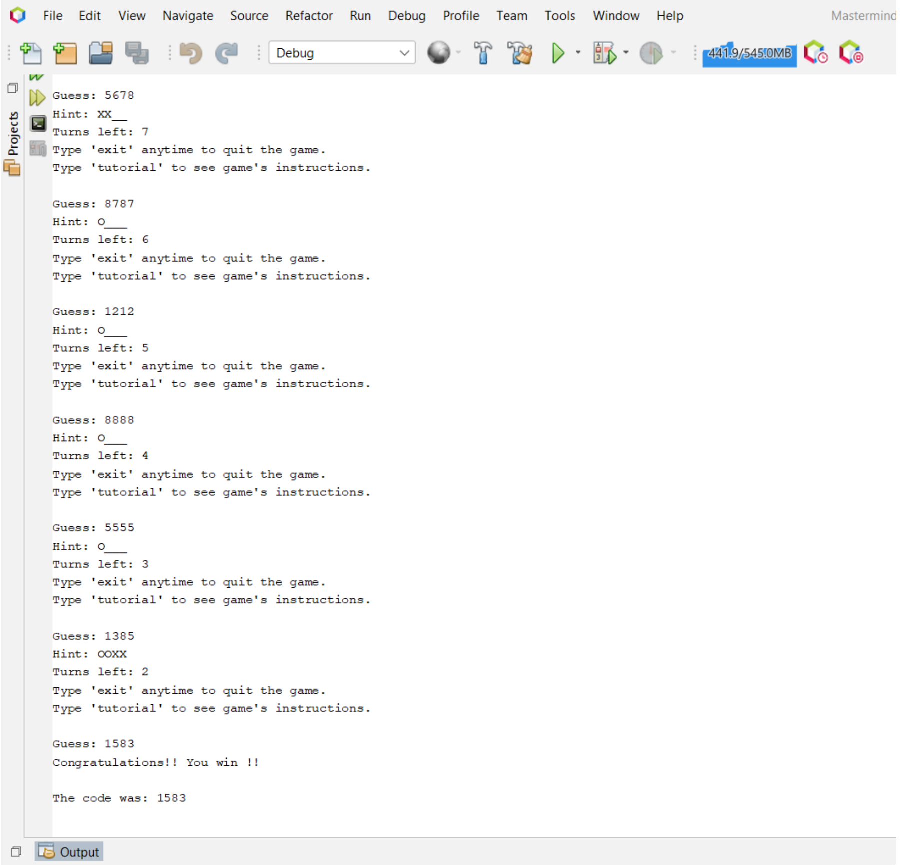
```
Guess: 786122
Hint: XX____
Turns left: 8
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 666666
Hint: OO____
Turns left: 7
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 665555
Hint: XXXX__
Turns left: 6
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 571466
Hint: OOXX__
Turns left: 5
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 157566
Hint: OOXX__
Turns left: 4
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 415766
Hint: OXXX__
Turns left: 3
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 752466
Hint: OOXX__
Turns left: 2
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.
```



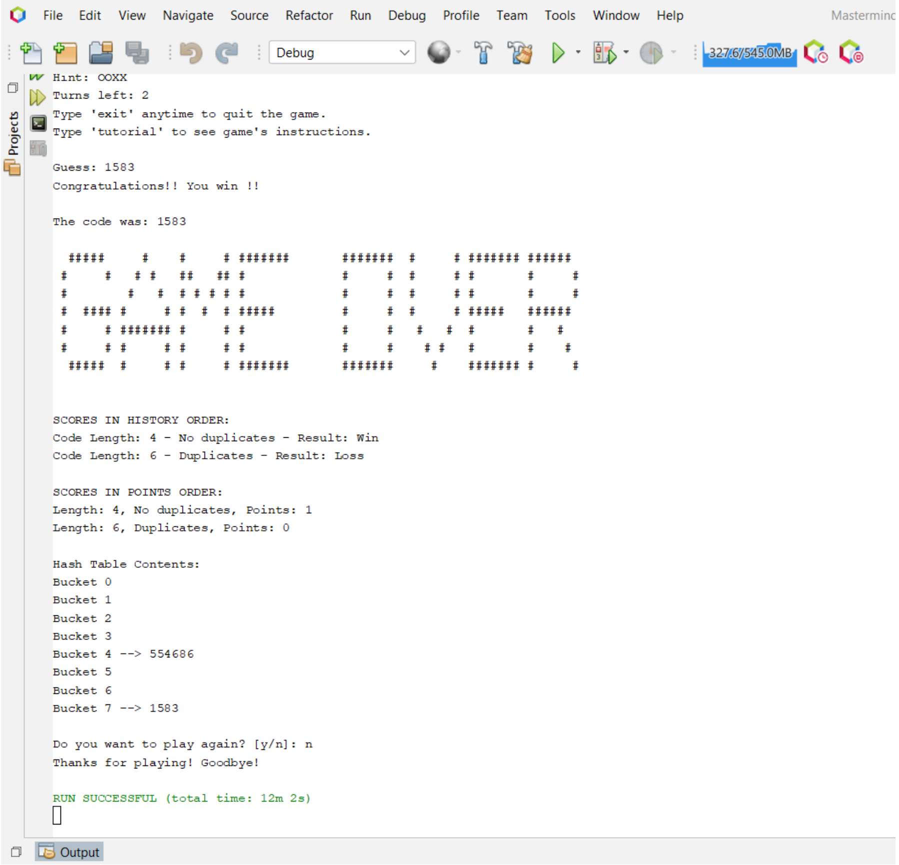
```
Guess: 752466
Hint: OOXX__
Turns left: 2
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 554466
Hint: OOOOX_
Turns left: 1
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 557466
Hint: OOOXX_
Turns left: 0

The code was: 554686

  #####     #    #     # #######       #######  #     # ####### ######
 #     #   # #   ##   ## #             #     #  #     # #       #     #
 #        #   #  # # # # #             #     #  #     # #       #     #
 #  #### #     # #  #  # #####         #     #  #     # #####   ######
 #     # ####### #     # #             #     #   #   #  #       #   #
 #     # #     # #     # #             #     #    # #   #       #    #
  #####  #     # #     # #######       #######     #    ####### #     #


SCORES IN HISTORY ORDER:
Code Length: 6 - Duplicates - Result: Loss

SCORES IN POINTS ORDER:
Length: 6, Duplicates, Points: 0

Hash Table Contents:
Bucket 0
Bucket 1
Bucket 2
Bucket 3
Bucket 4 --> 554686
Bucket 5
Bucket 6
```



```
Hash Table Contents:
Bucket 0
Bucket 1
Bucket 2
Bucket 3
Bucket 4 --> 554686
Bucket 5
Bucket 6
Bucket 7

Do you want to play again? [y/n]: y
Choose the code length:
4
6
8
4
Error: Invalid input type. Please enter a number.
Choose the code length:
4
6
8
4
Do you want to play with duplicates? [y/n]: n

Write a code using the numbers from 1 to 8. You have 10 turns to guess the code.
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 1234
Hint: OX__
Turns left: 9
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 3333
Hint: O___
Turns left: 8
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 5678
```



```
Guess: 5678
Hint: XX__
Turns left: 7
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 8787
Hint: O___
Turns left: 6
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 1212
Hint: O___
Turns left: 5
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 8888
Hint: O___
Turns left: 4
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 5555
Hint: O___
Turns left: 3
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 1385
Hint: OOXX
Turns left: 2
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 1583
Congratulations!! You win !!

The code was: 1583
```



```
Hint: OOXX
Turns left: 2
Type 'exit' anytime to quit the game.
Type 'tutorial' to see game's instructions.

Guess: 1583
Congratulations!! You win !!

The code was: 1583

  #####     #    #     # #######       ####### #     # ####### ######
 #     #   # #   ##   ## #             #     # #     # #       #     #
 #        #   #  # # # # #             #     # #     # #       #     #
 #  #### #     # #  #  # #####         #     # #     # #####   ######
 #     # ####### #     # #             #     #  #   #  #       #   #
 #     # #     # #     # #             #     #   # #   #       #    #
  #####  #     # #     # #######       #######    #    ####### #     #


SCORES IN HISTORY ORDER:
Code Length: 4 - No duplicates - Result: Win
Code Length: 6 - Duplicates - Result: Loss

SCORES IN POINTS ORDER:
Length: 4, No duplicates, Points: 1
Length: 6, Duplicates, Points: 0

Hash Table Contents:
Bucket 0
Bucket 1
Bucket 2
Bucket 3
Bucket 4 --> 554686
Bucket 5
Bucket 6
Bucket 7 --> 1583

Do you want to play again? [y/n]: n
Thanks for playing! Goodbye!

RUN SUCCESSFUL (total time: 12m 2s)
```

References

1. Dr. Lehr’s github. https://github.com/ml1150258/CSC_7
2. Gaddis, Tony. *Starting Out With C++: From Control Structures through Objects.* 7th ed., Pearson, 2012.
3. Mastermind - Rules of the game. https://webgamesonline.com/mastermind/rules.php
4. Standard Template Library Programmer's Guide. http://209.129.8.7/~MarkLehrSyllabi/sgi-stl-docs/docs/
5. C++ STL Tutorial. https://www.geeksforgeeks.org/cpp-stl-tutorial/

Program

```
/*************************************************
* Author    : Bryan Estrada                      *
* Teacher   : Dr. Mark Lehr                      *
* Class     : CSC-17C                            *
* Assignment: Project #2                         *
* Title     : Mastermind with Recursion and Tree *
*************************************************/

//Libraries
#include <iostream>
#include <cstdlib>   // Random Function Library
#include <ctime>     // Time Library
#include <string>
#include <set>
#include <list>
#include <stack>
#include <queue>
#include <map>
#include <vector>
#include <utility>
#include <algorithm>
using namespace std;

/************************************************************
*    Holds the result of a single game of Mastermind,
*    including game settings and outcome.
 ***********************************************************/
struct GameResult {
    int codeLength;
    char duplicateSetting;
    bool isWin; // true if user won, false if lost
    GameResult(int len, char dup, bool win)  {
        codeLength = len;
        duplicateSetting = dup;
        isWin = win;
    }
};

/************************************************************
*    A struct representing a node in a binary search tree,
*    where each node stores a game result and has pointers
*    to its left and right children to store and organize
*    game results for efficient retrieval and manipulation.
************************************************************/
struct TreeNode {
    GameResult result;
    TreeNode* left;
    TreeNode* right;
    TreeNode(GameResult gr) : result(gr){
        this->left = nullptr;
        this->right = nullptr;
    }
};

//Function prototypes
void setupGame();
char getDuplicateChoice();
int getCodeLength();
void genCode(int, list<char>&, char);
void genNums(int, list<char>&, deque<char>&, set<char>&, char);
void printCode(const list<char>&);
void hint(const list<char>&, const list<char>&);
void hint_recursive(const list<char>&, const list<char>&, map<char, int>&, int,
                    int, auto, auto);
void hint_recursive_misplaced(const list<char>&, const list<char>&,
                              map<char, int>&, int, int, auto, auto);
void showGameOverMessage(const list<char>&);
void showInstructions();
void validInput(const string&, bool&, const int&);
void compareGuess(list<char>&, const string&, const list<char>&, bool&,
                  stack<int>&, const int&, const char&, TreeNode*&);
void exitingGame(bool&);
void newGame(char&);
void recordResult(int, char, bool, TreeNode*&);
void displayStatistics(TreeNode*);
void printWelcome();
void printGameOver();
void insert(TreeNode*&, GameResult);
void printInOrder(TreeNode*);
void extractScores(TreeNode*, vector<pair<string, int>>&);
void merge(vector<pair<string, int>>&, int, int, int);
void mergeSort(vector<pair<string, int>>&, int, int);
void printSortedScores(TreeNode*);
unsigned int RSHash(const list<char>);

/************************************************************
*    A Hash Table implementation using chaining for collision
*    resolution, designed to handle keys represented as
*    `list<char>` objects.
************************************************************/
class HashTable {
private:
    vector<list<list<char>>> table; // Array of linked lists for chaining
    int size;                   // Number of buckets
    // Compute the index using RSHash and modulus operation
    int computeIndex(const list<char> key) const {
        return RSHash(key) % size;
    }

public:
    // Constructor
    HashTable(int tableSize) {
        this->size = tableSize;
        table.resize(size);
    }

    // Get the size of the hash table
    int getSize() const {
        return size;
    }

    // Get a bucket at a specific index
    const list<list<char>>& getBucket(int index) const {
        return table[index];
    }

    // Insert a key into the hash table
    void insert(const list<char>& key) {
        int index = computeIndex(key);
        table[index].push_back(key);
    }
        // Search for a key in the hash table
    bool search(const list<char>& key) const {
        int index = computeIndex(key);
        for (const auto &val : table[index]) {
            if (val == key) {
                return true;
            }
        }
        return false;
    }
};

void printHashTable(const HashTable&);

int main()
{
    queue<GameResult> resultsQueue;
    char playAgain = 'y';
    list<char> code;
    list<char> guess;
    char choiceDuplicate;
    int length;
    stack<int> turns;
    const int numTurns = 10;
    string guess_input;
    bool quit = false;
    TreeNode* resultsTree = nullptr;
    const int tableSize = 8;
    HashTable hashTable(tableSize);

    setupGame();    //Setting up the random function
    printWelcome();

    do {
        bool endGame = false;
        bool skipTurn = false; // Flag to skip the turn without using `continue`
        playAgain = tolower(playAgain);

        if(playAgain == 'y') {
            for (int i = 1; i <= numTurns; i++){
                turns.push(i);
            }

            // Get valid code length
            length = getCodeLength();

            // Get valid choice for duplicates
            choiceDuplicate = getDuplicateChoice();

            genCode(length, code, choiceDuplicate);
            hashTable.insert(code);
//            cout << "\t\tCODE: ";
//            printCode(code);
            cout << "\nWrite a code using the numbers from 1 to 8. You have 10 "
                    "turns to guess the code.\n";

            while (!endGame && !turns.empty() && !quit) {
                skipTurn = false; // Reset skipTurn flag at the start of each turn
                cout << "Type 'exit' anytime to quit the game." << endl;
                cout << "Type 'tutorial' to see game's instructions." << endl;
                cout << "\nGuess: ";
                cin >> guess_input;

                // Check for exit command
                if (guess_input == "exit") {
                    exitingGame(quit);
                    skipTurn = true; // Skip rest of the loop for re-confirmation
                }

                if (guess_input == "tutorial") {
                    showInstructions();
                    skipTurn = true;
                }

                // First try-catch block: Check guess input
                if(!skipTurn){
                    validInput(guess_input, skipTurn, length);
                }

                // Clear the previous guess and add the new one from input
                if(!skipTurn){
                    compareGuess(guess, guess_input, code, endGame, turns,
                                 length, choiceDuplicate, resultsTree);
                }
            }

            if (!quit) {
                showGameOverMessage(code);
                displayStatistics(resultsTree);
                printSortedScores(resultsTree);  //Statistics after each game
                printHashTable(hashTable);
                newGame(playAgain);
            }
        }
        code.clear();
    } while (playAgain == 'y' && !quit);

    return 0;
}

/************************************************************
*    Generates a random code for the Mastermind game based on
*    the specified length and duplicate setting.
 ***********************************************************/
void genCode(int length, list<char>& code, char choice) {
    set<char> unique_numbers;
    deque<char> numbers = {'1', '2', '3', '4', '5', '6', '7', '8'};

    genNums(length, code, numbers, unique_numbers, choice);
}

/************************************************************
*    Generates a random sequence of numbers for a Mastermind
*    game code using recursion. Handles both
*    duplicate-allowed and duplicate-restricted scenarios.
***********************************************************/
void genNums(int length, list<char>& code, deque<char>& numbers,
             set<char>& unique_numbers, char choice) {
    // Base case: If the desired length is reached, stop recursion.
    if (code.size() == length) {
        return;
    }

    // Shuffle the deque to randomize the selection.
    random_shuffle(numbers.begin(), numbers.end());

    // Get the first number from the shuffled deque.
    char num = numbers.front();

    // Check if duplicates are allowed or the number is unique.
    if (toupper(choice) == 'Y' || unique_numbers.find(num) == unique_numbers.end()) {
        code.push_back(num);
        unique_numbers.insert(num);
    }

    // Recursive call with reduced length.
    genNums(length, code, numbers, unique_numbers, choice);
```

```
}

/*************************************************************
*    Displays the generated code sequence by printing each
*    character in the code list.
 ***********************************************************/
void printCode(const list<char>& code) {
    for (char num : code) {
        cout << num;
    }
    cout << endl;
}

/*************************************************************
*    Generates and processes a hint for a Mastermind game
*    guess using recursion. The function calculates and
*    tracks the number of digits in the correct position
*    (`correct`) and unmatched code digits for later
*    misplaced digit analysis. Delegates misplaced digit
*    analysis to `hint_recursive_misplaced`.
************************************************************/
void hint_recursive(const list<char>& code, const list<char>& guess,
                    map<char, int>& code_count, int correct, int misplaced,
                    auto code_it, auto guess_it) {
    if (code_it == code.end()) {
        // Base case
        guess_it = guess.begin();
        code_it = code.begin();
        hint_recursive_misplaced(code, guess, code_count, correct, misplaced,
                                 code_it, guess_it);
        return;
    }

    // Process one element: Check if the current code and guess match at the same
position
    if (*code_it == *guess_it) {
        correct++;  // Increment correct count for each matching position
    } else {
        code_count[*code_it]++;  // Track unmatched code digits for misplaced
checking
    }

    // Move to the next characters
    hint_recursive(code, guess, code_count, correct, misplaced, ++code_it,
++guess_it);
}

/*************************************************************
*    Generates and prints a hint for a Mastermind game guess
*    using recursion. The hint shows:
*      - 'O' for correct digits in correct positions.
*      - 'X' for correct digits in incorrect positions.
*      - '_' for incorrect digits.
************************************************************/
void hint_recursive_misplaced(const list<char>& code, const list<char>& guess,
                              map<char, int>& code_count, int correct,
                              int misplaced, auto code_it, auto guess_it) {
    if (guess_it == guess.end()) {
        // Base case: If we've processed all guesses, print the hint result
        string hint_result(correct, 'O');   // Add all 'O's for correct positions
        hint_result += string(misplaced, 'X'); // Add all 'X's for misplaced digits
        hint_result += string(code.size() - correct - misplaced, '_'); // Add all
'_'s for incorrect digits
        cout << "Hint: " << hint_result << endl;
        return;
    }

    // Check for misplaced digits
    if (*code_it != *guess_it) {
        auto found = find(code.begin(), code.end(), *guess_it); // Search for the
guess in the rest of the code
        if (code_count[*found] > 0) {
            misplaced++; // Increment misplaced count if digit is in the wrong
position
            code_count[*found]--; // Decrement count to avoid double-counting
        }
    }

    // Move to the next characters
    hint_recursive_misplaced(code, guess, code_count, correct, misplaced,
                             ++code_it, ++guess_it);
}

/*************************************************************
*    Generates a hint to guide the player by indicating
*    the number of correct and misplaced digits in the guess.
 ************************************************************/
void hint(const list<char>& code, const list<char>& guess) {
    map<char, int> code_count;
    int correct = 0;     // Counts correct positions (O's)
    int misplaced = 0;   // Counts misplaced digits (X's)

    // Start recursive function to process code and guess
    hint_recursive(code, guess, code_count, correct, misplaced, code.begin(),
                   guess.begin());
}

/*************************************************************
*    Initializes the random number generator with the current
*    time to ensure different random sequences in each game.
 ************************************************************/
void setupGame(){
    srand(static_cast<unsigned int>(time(0)));
}

/*************************************************************
*    Prompts the user to select a code length for the game
*    and validates the input, ensuring it is 4, 6, or 8.
 ************************************************************/
int getCodeLength(){
    int length;

    do {
        try {
            cout << "Choose the code length: " << endl;
            cout << "4" << endl;
            cout << "6" << endl;
            cout << "8" << endl;
            cin >> length;
            if (cin.fail()){
                throw invalid_argument("Invalid input type. Please enter a number.");
            }
            if (length != 4 && length != 6 && length != 8){
                throw invalid_argument("Invalid code length. Please enter 4, 6, or
8.");
            }
        }
        catch (const invalid_argument& e) {
            cout << "Error: " << e.what() << endl;
            cin.clear();
            cin.ignore(100, '\n');
            length = 0;
        }
    } while (length != 4 && length != 6 && length != 8);

    return length;
}

/*************************************************************
*    Prompts the user to decide if duplicates are allowed in
*    the game code, validating the input as either 'y' or
*    'n'.
 ***********************************************************/
char getDuplicateChoice(){
    char choiceDuplicate;

    do {
        try {
            cout << "Do you want to play with duplicates? [y/n]: ";
            cin >> choiceDuplicate;
            if (cin.fail()){
                throw invalid_argument("Invalid input type. Please enter 'y' or
'n'.");
            }
            choiceDuplicate = tolower(choiceDuplicate);
            if (choiceDuplicate != 'y' && choiceDuplicate != 'n'){
                throw invalid_argument("Invalid choice. Please enter 'y' or 'n'.");
            }
        }
        catch (const invalid_argument& e) {
            cout << "Error: " << e.what() << endl;
            cin.clear();
            cin.ignore(100, '\n');
            choiceDuplicate = '\0';
        }
    } while (choiceDuplicate != 'y' && choiceDuplicate != 'n');

    return choiceDuplicate;
}

/************************************************************
*    Displays the end-of-game message, reveals the correct
*    code, and displays a game over message.
 ***********************************************************/
void showGameOverMessage(const list<char> &code){
    cout << "\nThe code was: ";
    printCode(code);
    printGameOver();
}

/************************************************************
*    Ask the user if he or she wants to play again. If the
*    the user responds 'y', the game will start over with a
*    new secret code. If 'n', the program ends.
 ***********************************************************/
void newGame(char &playAgain){
    cout << "\nDo you want to play again? [y/n]: ";
    cin >> playAgain;
    playAgain = tolower(playAgain);
    if (playAgain == 'n') {
        cout << "Thanks for playing! Goodbye!" << endl;
    }
}

/************************************************************
*    Displays the instructions for the Mastermind game,
*    explaining the rules, the goal of the game, how guesses
*    and hints work, and how to enter valid inputs.
 ***********************************************************/
void showInstructions(){
    for(int i = 0 ; i < 80; i++){
        cout << "*";
    }

    cout << endl;
    cout << "*\t\t\tThis is Mastermind!" << endl << "*" << endl;
    cout << "*\tThe goal of the game is to guess the code the computer generated.";
    cout << endl << "*" << endl;
    cout << "*\tYou have 10 attempts to guess the code." << endl;
    cout << "*\tIn order to enter your guess, please type numbers from 1 to 8, "
            "\n*\taccording to the code size you selected (4, 6 or 8 digits).";
    cout << endl << "*" << endl;
    cout << "*\tFor every guess you entered, you will be given a hint in the form:";
    cout << endl << "*" << endl;
    cout << "*\tOOX_" << endl << "*" << endl;
    cout << "*\tThe symbols above represents the amount of digits in the right "
            "\n*\tposition, wrong position and incorrect digits from your guess: ";
    cout << endl;
    cout << "*\tO: One digit in the right position." << endl;
    cout << "*\tX: One digit in the wrong position. " << endl;
    cout << "*\t_: One incorrect digit." << endl << "*" << endl;
    cout << "*\tFor instance, if the secret code is '1234' and your guess was "
            "\n*\t'5247', the hint will be OX__, because '2' was in the right "
            "\n*\tposition, '4' was in the wrong position and '5' and '7' were "
            "\n*\tincorrect digits. As you can notice, the hint does not show you "
            "\n*\twhat digit's place was right, wrong or incorrect, it only shows "
            "\n*\tthe amount." << endl << "*" << endl;
    cout << "*\t\t\tHAPPY GUESSING! :D" << endl;

    for(int i = 0 ; i < 80; i++){
        cout << "*";
    }
    cout << endl;
}

/************************************************************
*    Validates the player's guess input for correctness
*    in terms of format, length, and valid characters (1-8).
 ***********************************************************/
void validInput(const string &guess_input, bool &skipTurn, const int &length){
    try {
        if (guess_input.empty()){
            throw invalid_argument("Input cannot be empty. Please try again.");
        }
        if (!all_of(guess_input.begin(), guess_input.end(), ::isdigit)){
            throw invalid_argument("Guess contains invalid characters. Use only 
numbers.");
        }
        if (guess_input.size() != length){
            throw invalid_argument("Guess length does not match the code length.");
        }
        for (char ch : guess_input) {
            if (ch < '1' || ch > '8')
                throw invalid_argument("Guess contains invalid numbers. Only use 1 to 
8.");
        }
    } catch (const invalid_argument& e) {
        cout << "Error: " << e.what() << endl;
        skipTurn = true; // Skip turn if an invalid guess was made
    }
}

/************************************************************
*    Compares the player's guess to the generated code,
*    provides feedback through hints, and determines if the
*    game is won or lost.
 ***********************************************************/
void compareGuess(list<char>& guess, const string& guess_input,
                  const list<char>& code, bool& endGame, stack<int>& turns,
                  const int &length, const char &choiceDuplicate,
                  TreeNode*& resultsTree) {  // TreeNode* instead of queue
    guess.clear();
    for (char ch : guess_input){
        guess.push_back(ch);
    }
    if (code == guess) {
        endGame = true;
        recordResult(length, choiceDuplicate, true, resultsTree); // Record win
        cout << "Congratulations!! You win !!" << endl;
        while(!turns.empty()){
            turns.pop();
        }
    } else {
        hint(code, guess);
        if (!turns.empty()) {
            turns.pop();
            cout << "Turns left: " << (turns.empty() ? 0 : turns.top()) << endl;
            if(turns.empty()){
                recordResult(length, choiceDuplicate, false, resultsTree); // Record
loss
            }
        }
    }
}

/************************************************************
*    Asks the player for confirmation to exit the game and
*    sets the quit flag if the player confirms.
 ***********************************************************/
void  exitingGame(bool &quit){
    char confirm;
    cout << "Are you sure you want to quit? [y/n]: ";
    cin >> confirm;
    if (tolower(confirm) == 'y') {
        cout << "Exiting game. Thanks for playing!" << endl;
        quit = true; // Set quit flag to exit loop
    }
}

/************************************************************
*    Records the outcome of a single game (win/loss) and its
*    settings into a queue for tracking game results.
 ***********************************************************/
void recordResult(int codeLength, char duplicateSetting, bool isWin,
                  TreeNode*& resultsTree) {  // TreeNode* instead of queue
    GameResult gr(codeLength, duplicateSetting, isWin);
    insert(resultsTree, gr); // Insert into the tree
}

/************************************************************
*    Displays the statistics of the game results, including
*    wins and losses for different code lengths (4, 6, 8) and
*    settings for duplicates, and compares the number of wins
*    with and without duplicates.
 ***********************************************************/
void displayStatistics(TreeNode* resultsTree) {  // TreeNode* instead of queue
    cout << "\nSCORES IN HISTORY ORDER:" << endl;
    printInOrder(resultsTree);  // Print tree in sorted order
    cout << endl;
```

```
}

/************************************************************
*    Create a visually striking title screen for the game.
 ***********************************************************/
void printWelcome(){
    cout << endl;
    cout << "             (           )    *                   )      *      "
            "     (            (      *    (      ) (      " << endl;
    cout << " (  (         )\\ )  (  ( /(  (  `          *   ) ( /(    (  `    ("
            "     )\\ ) *   )    )\\ ) (  `   )\\ ) ( /( )\\ )   " << endl;
    cout << " )\\))(   '(  (()/(  )\\ )\\()) )\\))(  (    ` )  /( )\\())   )\\))(
)"
            "\\   (()/` )  /((  (()/( )\\))( (()/( )\\()(()/(   " << endl;
    cout << "((_)()\\ ) )\\  /(_)(((_((_)\\ ((_)()\\ )\\    ( )(_)((_)\\
((_)()(((("
            "_)(  /(_)( )(_))\\  /(_)((_)()\\ /(_)((_)\\ /(_))  " << endl;
    cout << "_(())\\_)(((_)(_)) )\\___ ((_)(_()((_((_)  (_(_())  ((_)  (_()((_)\\ "
            "_ )\\(_))(_(_()((_)(_)) (_()((_(_))  _((_(_))_   " << endl;
    cout << "\\ \\((_)/ | __| | ((/ __/ _ \\|  \\/  | __| |_   _| / _ \\  |  \\/
(_)"
            "_\\(_/ __|_   _| __| _ \\|  \\/  |_ _|| \\| ||   \\  " << endl;
    cout << " \\ \\/\\/ /| _|| |__| (_| (_) | |\\/| | _|    | |  | (_) | | |\\/| |/ "
            "_ \\ \\__ \\ | | | _||   /| |\\/| || | | .` || |) | " << endl;
    cout << "  \\_/\\_/ |___|____|\\___\\___/|_|  |_|___|   |_|   \\___/  |_|  |_/_/"
            " \\_\\|___/ |_| |___|_|_\\|_|  |_|___||_|\\_||___/  " << endl << endl;
}

/************************************************************
*    Create a ASCII art-style game over message.
 ***********************************************************/
void printGameOver(){
    cout << endl;
    cout << "  #####     #    #     # #######       #######  #     # ####### ######
" << endl;
    cout << " #     #   # #   ##   ## #             #     #  #     # #       #     #
" << endl;
    cout << " #        #   #  # # # # #             #     #  #     # #       #     #
" << endl;
    cout << " #  #### #     # #  #  # #####         #     #  #     # #####   ######
" << endl;
    cout << " #     # ####### #     # #             #     #   #   #  #       #   #
" << endl;
    cout << " #     # #     # #     # #             #     #    # #   #       #    #
" << endl;
    cout << "  #####  #     # #     # #######       #######     #    ####### #     #
" << endl;
    cout << endl;
}

/************************************************************
*    Inserts a `GameResult` object into a binary search tree
*    based on the code length and duplicate setting.
***********************************************************/
void insert(TreeNode*& root, GameResult gr) {
    if (root == nullptr) {
        root = new TreeNode(gr);
    } else if (gr.codeLength < root->result.codeLength ||
               (gr.codeLength == root->result.codeLength && gr.duplicateSetting <
root->result.duplicateSetting)) {
        insert(root->left, gr);
    } else {
        insert(root->right, gr);
    }
}

void printInOrder(TreeNode* root) {
    if (root != nullptr) {
        printInOrder(root->left);
        cout << "Code Length: " << root->result.codeLength << " - ";
        cout << (root->result.duplicateSetting == 'y' ? "Duplicates" : "No
duplicates");
        cout << " - Result: " << (root->result.isWin ? "Win" : "Loss") << endl;
        printInOrder(root->right);
    }
}

/************************************************************
*    Recursively traverses a binary tree to extract game
*    scores and their associated details into a vector of
*    pairs. This function prepares scores for further
*    processing, such as sorting or display.
***********************************************************/
void extractScores(TreeNode* root, vector<pair<string, int>>& scores) {
    if (!root) return;

    // Traverse the left subtree
    extractScores(root->left, scores);

    // Process the current node: Convert GameResult to a score format
    string key = "Length: " + to_string(root->result.codeLength) + ", " +
                 (root->result.duplicateSetting == 'y' ? "Duplicates" : "No
duplicates");
    int value = root->result.isWin ? 1 : 0; // Example scoring: 1 for a win, 0 for a
loss
    scores.emplace_back(key, value);

    // Traverse the right subtree
    extractScores(root->right, scores);
}

void merge(vector<pair<string, int>>& scores, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<pair<string, int>> leftArr(scores.begin() + left, scores.begin() + mid +
1);
    vector<pair<string, int>> rightArr(scores.begin() + mid + 1, scores.begin() +
right + 1);
```

```
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i].second >= rightArr[j].second) {
            scores[k++] = leftArr[i++];
        } else {
            scores[k++] = rightArr[j++];
        }
    }

    while (i < n1) scores[k++] = leftArr[i++];
    while (j < n2) scores[k++] = rightArr[j++];
}

void mergeSort(vector<pair<string, int>>& scores, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(scores, left, mid);
        mergeSort(scores, mid + 1, right);
        merge(scores, left, mid, right);
    }
}

/*************************************************************
*    Extracts and displays game scores from a binary tree
*    in sorted order. Provides a clear overview of scores
*    ranked by performance.
************************************************************/
void printSortedScores(TreeNode* root) {
    vector<pair<string, int>> scores;

    // Extract scores from the tree
    extractScores(root, scores);

    // Sort the scores
    mergeSort(scores, 0, scores.size() - 1);

    // Print the sorted scores
    cout << "SCORES IN POINTS ORDER:\n";
    for (const auto& score : scores) {
        cout << score.first << ", Points: " << score.second << endl;
    }
}

unsigned int RSHash(const list<char> str)
{
   unsigned int b    = 378551;
   unsigned int a    = 63689;
   unsigned int hash = 0;

   for(auto i : str)
   {
      hash = hash * a + i;
      a    = a * b;
   }
   return hash;
}

/************************************************************
*    Displays the contents of a hash table, bucket by bucket.
*    The output format makes the structure and contents of
*    the hash table clear and easy to understand.
***********************************************************/
void printHashTable(const HashTable& hashTable) {
    cout << "\nHash Table Contents:" << endl;
    for (int i = 0; i < hashTable.getSize(); ++i) {
        cout << "Bucket " << i;
        auto &bucket = hashTable.getBucket(i); // Access the bucket at index `i`
        if (!bucket.empty()) {
            cout << " --> ";
        }
        int j = 0;
        for (const auto& innerList : bucket) {
            for (char num : innerList) {
                cout << num;
            }
            j++;
            if(j != bucket.size()){
                cout << " --> ";
            }
        }
        cout << endl;
    }
}
```