**Documentation of Treasure\_Hunt\_game development and prompts used.**

#### ****1. Conceptualization of the Game****

The idea behind the game was to create a simple but engaging **"Treasure Hunt"** game where players guess the coordinates of hidden treasures on a 10×10 grid. I aimed to include interactive gameplay mechanics, visual feedback, and a dynamic hint system. The game needed to be easy to play while providing hints to guide the player to find the treasures. Here's the process I followed to bring the concept to life.

### ****Development Process****

#### ****Step 1: Choosing the GUI Library****

To create a user-friendly interface, I decided to use **PyQt5**, a Python library for building graphical user interfaces (GUIs). This library allowed me to create an interactive grid with clickable buttons, essential for the game's mechanics.

* **Reason for Using PyQt5**:  
  PyQt5 provides a flexible and customizable interface to handle visual components like grids, buttons, and labels, making it an ideal choice for developing this game.

#### ****Step 2: Setting Up the Grid****

I needed a grid layout to display the 10×10 squares where the treasures would be hidden. Each square was represented by a button that the player could click to make guesses. Using **QGridLayout**, I placed 100 buttons in a 10×10 format. Each button is associated with coordinates that correspond to its position on the grid.

#### ****Step 3: Treasure Placement****

To make the game dynamic, I used Python’s random module to randomly place 5 treasures across the grid. This ensures a unique game experience for every player.

**Code Implementation**:

def generate\_treasures(self):

while len(self.treasures) < self.total\_treasures:

x = random.randint(0, 9)

y = random.randint(0, 9)

if (x, y) not in self.treasures:

self.treasures.append((x, y))

This function generates random coordinates for each treasure and ensures there are no duplicate treasure locations.

#### ****Step 4: Guessing and Feedback System****

The core of the game involves players guessing the location of the hidden treasures. I implemented this by assigning a function to each button that checks the coordinates of the clicked square against the treasure locations.

* **Hot and Cold Hint System**:  
  After each guess, the game calculates the distance between the guessed location and the nearest treasure. If the guess is within 2 squares, the player receives a "Hot" hint; otherwise, the guess is marked "Cold."
* **Code for Feedback**:

def check\_guess(self, x, y):

if (x, y) in self.treasures:

self.feedback\_label.setText(f"Treasure Found at ({x+1}, {y+1})!")

self.buttons[(x, y)].setStyleSheet("background-color: gold")

else:

distances = [abs(x - tx) + abs(y - ty) for tx, ty in self.treasures]

min\_distance = min(distances)

if min\_distance <= 2:

self.feedback\_label.setText("Hot!")

self.buttons[(x, y)].setStyleSheet("background-color: red")

else:

self.feedback\_label.setText("Cold!")

self.buttons[(x, y)].setStyleSheet("background-color: blue")

#### ****Step 5: Visual Feedback****

For enhanced user experience, I added colour-coded feedback:

* **Gold for treasures** found.
* **Red for "Hot" guesses** (close to treasure).
* **Blue for "Cold" guesses** (far from treasure).

#### ****Step 6: Game Completion and Score Tracking****

Once all treasures are found, the game congratulates the player and displays the total number of guesses. This feature adds a scoring element, encouraging players to improve their guess count on subsequent plays.

* **Game Completion Logic**:

if self.found\_treasures == self.total\_treasures:

self.feedback\_label.setText(f"Congratulations! You found all treasures in {self.guesses} guesses!")

### ****3. Prompts Used in the Game Creation****

#### ****Initial Game Concept****:

* **Prompt**: "Create a simple guessing game where players try to find hidden treasures on a 10×10 grid. Include a hint system that tells the player if their guess is near or far from a treasure."
  + **Response**: This prompt helped lay the foundation for the gameplay mechanics, leading to the idea of a **Hot and Cold** hint system based on proximity.

#### ****Randomized Treasure Placement****:

* **Prompt**: "How can I randomly place treasures on a grid without overlapping?"
  + **Response**: The suggestion to use the random.randint() function and check for duplicate coordinates was crucial to ensure unique treasure locations.

#### ****Feedback and Visuals****:

* **Prompt**: "How can I visually represent the treasure locations and feedback for correct and incorrect guesses?"
  + **Response**: This led to the idea of using colors to differentiate between hot, cold, and treasure-found guesses. The use of **QPushButton** style changes (like setStyleSheet) allowed for a more engaging visual experience.

#### ****Game Completion and Scoring****:

* **Prompt**: "How can I track the player's progress and give feedback when all treasures are found?"
  + **Response**: I implemented a counter that tracks the number of treasures found and displays a congratulatory message when all treasures are discovered.

### ****Challenges Faced and Solutions****

1. **Issue**: Ensuring that the randomly placed treasures were unique.
   * **Solution**: I added a check to avoid placing treasures on the same coordinates twice.
2. **Issue**: Properly handling user input through clickable buttons.
   * **Solution**: I used lambda functions in button connections to pass coordinates as parameters.
3. **Issue**: Providing real-time feedback and color changes.
   * **Solution**: I used the setStyleSheet method of PyQt5’s QPushButton to change button colors dynamically based on player guesses.

**6. Conclusion**

The "Treasure Hunt" game combines interactive elements, user feedback, and a bit of strategy, creating an enjoyable experience for players. By leveraging AI prompts and incorporating random treasure placement and proximity-based hints, the game is both challenging and fun to play. This project also demonstrates how basic Python knowledge and GUI libraries like PyQt5 can be used to create engaging and interactive applications.

**Explanation of the code**

Importing necessary modules.

Now, let's define the TreasureHunt class, which inherits from QWidget. This class represents the main window of the game.

The \_\_init\_\_ method initializes the game by calling the initUI method.

The initUI method sets up the user interface of the game.

self.setWindowTitle("Treasure Hunt Game") sets the title of the game window.

self.grid = QGridLayout() creates a grid layout for the game board.

self.treasures = [] initializes an empty list to store the coordinates of the treasures.

self.guesses = 0 initializes a counter for the number of guesses.

self.found\_treasures = 0 initializes a counter for the number of treasures found.

self.total\_treasures = 5 sets the total number of treasures to find.

self.feedback\_label = QLabel("Guess the treasures!") creates a label to display feedback messages.

self.layout = QVBoxLayout() creates a vertical layout to arrange the grid and feedback label.

self.setLayout(self.layout) sets the layout of the game window.

Next, the initUI method initializes the game board with buttons.

self.buttons = {} initializes an empty dictionary to store the buttons.

The nested loops iterate over the 10x10 grid and create a button for each cell.

button.setFixedSize(50, 50) sets the size of the button to 50x50 pixels.

button.clicked.connect(lambda ch, x=x, y=y: self.check\_guess(x, y)) connects the button's clicked signal to the check\_guess method.

self.grid.addWidget(button, x, y) adds the button to the grid layout.

self.buttons[(x, y)] = button stores the button in the self.buttons dictionary.

After initializing the game board, the initUI method generates random treasures.

The generate\_treasures method generates random coordinates for the treasures.

while len(self.treasures) < self.total\_treasures: generates treasures until the desired number of treasures is found.

x = random.randint(0, 9) generates a random x-coordinate.

y = random.randint(0, 9) generates a random y-coordinate.

if (x, y) not in self.treasures: checks if the generated coordinates are already in the self.treasures list.

self.treasures.append((x, y)) adds the generated coordinates to the self.treasures list.

The check\_guess method is called when the user clicks on a button.

self.guesses += 1 increments the number of guesses.

if (x, y) in self.treasures: checks if the clicked button contains a treasure.

self.feedback\_label.setText(f"Treasure Found at ({x+1}, {y+1})! Total treasures found: {self.found\_treasures + 1}") displays a message indicating that a treasure was found.

self.buttons[(x, y)].setStyleSheet("background-color: gold") changes the background color of the button to gold.

self.buttons[(x, y)].setEnabled(False) disables the button.

self.found\_treasures += 1 increments the number of treasures found.

if self.found\_treasures == self.total\_treasures: checks if all treasures have been found.

self.feedback\_label.setText(f"Congratulations! You found all treasures in {self.guesses} guesses!")