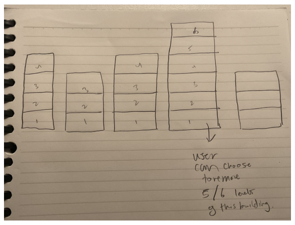
Towers of Nim project documentation

LINK TO VIDEO (Sound effects do not work)

1. Game Mechanics & how to play the game:

This game is dubbed "Towers of logic", and it is a unique concept that I came up with, inspired from the Jenga game. The logic of this game is simple: you start with a specific number of 'towers' which are assigned a random height (between 3 and 8). The first level starts with 5 towers. The user plays the first turn, and can choose to remove any number of 'levels' from any one 'tower'. An example of this is demonstrated below.



Next, when the 5 blocks from tower 4 are removed, a recursive pattern will be implemented to add height of 2 to the towers immediately to the right and left of tower 4, and then add 1 to the tower to the left of the tower to the left of tower 4 (tower 2), and the tower to the right of the tower to the right of block 4 (tower 6, but does not exist in this scenario) would also get 1 added to it's height. Hence the height representation of the towers would be as follows:

Tower number:	**1**	**2**	**3**	**4**	**5**
Initial heights	4	3	4	6	3
Final heights after move 1:	4 (no change, too far from tower 4)	4 (1 added)	6 (2 added)	1 (5 removed)	5 (2 added)

After the user plays, they press the 'computer turn' button, and a recursive algorithm implements the computer's turn, and a window is prompted stating how many rows the computer chose to remove and from what tower. Note that the code does not allow the user to play two consecutive turns, or for the computer to be prompted to play two consecutive turns either. The end goal is to be the first reason who causes one of the rows to achieve a length of higher than 15. Although this might sound easy, it can be quite tricky to achieve. The code is able to recognize who wins the game and announces it in a new window.

The game also has a twist of the 'Earthquake' feature, which can be used by the player only once in their turn, and basically what it does is recursively remove 2 from the length of all the towers. At every level, the player is only able to do it once.

Once rules are understood, the game is pretty straightforward. The goal is to be the first to be able to get a tower to length 15 or higher. If the player was to get lost at any point in the game, or wants a refresh of the rules, the top left section of the game window has a button, which will show the instructions when pressed. When the player wants to exit, they can just close the window and exit out of the game.

2. *Recursion Implementation and Evidence:*

1. implementEarthquake(int n)

- This method is recursively called to implement the "Earthquake" feature. The earthquake reduces the height of all towers by 2 levels.

- Logic:

    - If n == 0, it's the base case: It plays the earthquake sound and adjusts the height of the first tower. If the first tower's height is more than 2, it subtracts 2 from its height; otherwise, it sets the height to 1. The method then updates the tower panel to reflect this change.

    - If n != 0, it adjusts the height of the nth tower in the same way, then recursively calls itself with n-1.

- The recursion ensures that every tower gets affected by the earthquake.

2. possibleMoves()

- This method isn't recursive itself but is crucial for the recursive Minimax algorithm. It determines which towers are valid to have blocks removed from (i.e., towers with more than 1 block).

3. minimax(ArrayList<Tower> currentState, boolean isMaximizing, int depth)

- Minimax is a type of algorithm that is used in many games that basically predicts the next moves and goes through all the scenarios and plays the move that most benefits the computer. The maximizing player is the computer, and the minimizing player is the user(human).

- Each state of the game has a value. In this code:
    - A state where the computer wins is positive.
    - A state where the player wins is negative.
    - A neutral state (neither win nor lose) is zero.

- The algorithm works in two modes:

    - Maximizing Player (Computer): Tries to maximize the score. It looks for moves that lead to a win or at least the best possible outcome.
    - Minimizing Player (Human): Tries to minimize the score. It looks for moves that lead to a loss for the computer or the worst outcome for the computer.

- The Base Case checks if any tower's height is 15 or more. If yes, a value is returned based on which player's turn it is and how deep the recursion has gone.

- Another base case is checking the depth. If the depth is 0, the recursion stops. This acts as a boundary to prevent the algorithm from going too deep and taking too much time.

- If it's the maximizing player's turn (computer's turn), the algorithm checks all possible moves, applies each move, and then recursively calls itself assuming the player will then make their best move. The algorithm picks the move that gives the highest score.

- If it's the minimizing player's turn, the algorithm does the same but tries to minimize the score, assuming the computer will make its best move next.

Supporting Methods:

- possibleMoves(): This method returns a list of valid tower indices from which blocks can be removed. It's the starting point to explore all possible moves for the current game state.

- applyMove(ArrayList<Tower> currentState, int towerIndex, int player): This method simulates what the game state would look like after a given move. It doesn't alter the actual game state but returns a new hypothetical state.

- growAdjacentTowers(int towerIndex, int growthAmount, ArrayList<Tower> currentState): This method is responsible for adjusting the heights of towers adjacent to a given tower.

It's crucial because every move not only affects the selected tower but also the adjacent ones.

4. applyMove(ArrayList<Tower> currentState, int towerIndex, int player)

- This method simulates the effect of a move without actually applying it to the main game state. It's used by the Minimax algorithm to explore possible future game states.

- Logic:
    - It first creates a new state (newState) by copying the current state.
    - It then adjusts the height of the selected tower (towerIndex) and uses the overloaded growAdjacentTowers method to adjust adjacent towers.

5. growAdjacentTowers(int towerIndex, int growthAmount, ArrayList<Tower> currentState):

- This overloaded method adjusts the height of towers adjacent to a given tower in a hypothetical game state (used by the applyMove method).

- Logic:
    - It increases the height of the towers immediately adjacent to the selected tower by growthAmount.
    - The towers two places away from the selected tower have their heights increased by growthAmount - 1.

6. computerMove()

- This method uses the above recursive Minimax function to determine the computer's best move.

- Logic:

    - For each possible move, it applies that move, then uses the Minimax algorithm to evaluate the resulting game state.
    - The move that results in the highest value is chosen as the best move.

- createTowerButton(int index): This method is used to create GUI components for each tower recursively.-
    - I used this method because each different 'level' of the game would require a different number of buttons (essentially towers).

3. Data Structures/Text files being used:

- **Arraylists:**
    - private ArrayList<JButton> towerButtons = new ArrayList<>();
        - This ArrayList holds JButton objects, which correspond to the tower buttons displayed on the GUI.
        - Dynamic sizing.

    - private ArrayList<Tower> towers = new ArrayList<>();
        - This ArrayList holds Tower objects, which represent the state of each tower in terms of the number of blocks it has.

- **Primitive Data Types:**
    - int: Used for counters, indices, tower heights, and various numeric values (e.g., numTowers, the height of a tower in the Tower class, etc.).
    - boolean: Used for flags and conditions (e.g., isPlayerTurn, earthquakeTried).

- **Custom Class (Tower):**
    - The Tower class is a simple data structure used to encapsulate the properties and behaviors of a tower. It primarily stores the number of blocks (height) a tower has.
    - Here is the tower class:

```
private class Tower {
    int blocks;
    Tower(int blocks) {
        this.blocks = blocks;
    }
    public int getHeight() {
        return this.blocks;
    }
    private void setHeight(int height) {
        this.blocks = height;
```

- Has a getter and setter method, quite basic but helps make code a lot more understandable and also adds convenience.

```
    }
}
```

4. Success Criteria:

This journey with creating the recursion project was quite tough, as I realized that my initial idea had no scope to implement complex recursion. I decided to change my idea and create a better and more complex game that could implement recursion as well. One of my main objectives going into this project was that I wanted to create a computer playing algorithm that wasn't just based on randomness but had some logic behind it. Finding a way to put this in code was difficult, but I think that my computerMove() algorithm satisfies my initial vision. It isn't an unbeatable computer algorithm, because that would take the fun out of the game, but rather an algorithm that is smart and can both play offensively and defensively. The idea for the game did take a while to come up with, and I think that my implementation has hit my initial success criteria, and I am happy with my project. However, I felt that my efficiency was not the best, and at some times my recursion implementations were quite basic as well. On the contrary, I found my minimax recursive algorithm quite complex, and I am happy that I was able to implement multi level recursion, and it took me a long time to fine tune it and to get it to work.