**📘 Documentation of Minimax Algorithm Code**

**1. Overview**

This Python program implements the **Minimax Algorithm**, which is a **decision-making algorithm** used in **Artificial Intelligence**, especially in **two-player games** (like Tic-Tac-Toe, Chess, etc.).  
It assumes that one player tries to **maximize** the score while the other tries to **minimize** it.  
The algorithm explores all possible outcomes to find the **best possible move** for the maximizing player.

**2. Function: minmax(depth, is\_max, values, index, max\_depth)**

**Purpose:**  
This recursive function calculates the **optimal score** a player can achieve, assuming both players play optimally.

**Parameters:**

* depth: Current level of the game tree (how deep the recursion has gone).
* is\_max: Boolean value — True if it’s the **maximizer’s** turn, False if it’s the **minimizer’s** turn.
* values: List of leaf node values (final scores at the end of the game tree).
* index: Current index in the values list being evaluated.
* max\_depth: Maximum depth of the game tree (how many levels until the leaf nodes).

**Working:**

1. **Base Case:**  
   If depth == max\_depth, it means we reached a leaf node → return its value (values[index]).
2. **Recursive Calls:**
   * Calculate the left and right child values using recursive calls:
   * left = minmax(depth + 1, not is\_max, values, index \* 2, max\_depth)
   * right = minmax(depth + 1, not is\_max, values, index \* 2 + 1, max\_depth)
3. **Decision Step:**
   * If it’s the maximizer’s turn → choose the **maximum** of left and right.
   * If it’s the minimizer’s turn → choose the **minimum** of left and right.

**3. Function: find\_depth(values)**

**Purpose:**  
To determine how many levels (depth) the game tree has, based on the number of leaf nodes.

**Logic:**

* For a complete binary tree, the number of leaf nodes must be a **power of 2**.
* Example: if there are 8 leaf nodes → depth = 3 (since 2³ = 8).

**Steps:**

1. Start with count = 0.
2. Keep doubling 2 \*\* count until it equals or exceeds the number of values.
3. If it never equals exactly → return -1 (error).
4. Otherwise → return the depth count.

**4. Leaf Node Values**

leaf\_values = [3, 5, 2, 9, 3, 5, 2, 9]

These represent the **scores** at the terminal nodes of the game tree (end states).  
The algorithm will use these to decide the **best possible outcome**.

**5. Main Program Flow**

1. Find the depth of the tree using find\_depth(leaf\_values).
2. If the number of leaves is not a power of 2 → show error message.
3. Otherwise:
4. best\_score = minmax(0, True, leaf\_values, 0, depth)
5. print("Best possible score for the maximizing player is:", best\_score)

**6. Output**

Best possible score for the maximizing player is: 9

**7. Explanation of Result**

* The algorithm simulates all possible moves for both players.
* The **maximizer** tries to get the **highest possible value**, while the **minimizer** tries to reduce it.
* After exploring all paths, the **best guaranteed score** for the maximizer is **9**.

**8. Summary (in short)**

| **Concept** | **Explanation** |
| --- | --- |
| Algorithm | Minimax |
| Type | Recursive decision-making |
| Players | Maximizer & Minimizer |
| Purpose | To find best possible score assuming both play optimally |
| Base Case | When depth == max\_depth |
| Result | Best score for maximizer = 9 |