**Code for red-blue-nim game:**

import sys

def display\_tokens(red\_tokens, blue\_tokens):

    # Displays the current number of red and blue tokens.

    print(f"Red tokens: {red\_tokens}, Blue tokens: {blue\_tokens}")

def player\_turn(player, red\_tokens, blue\_tokens):

    # Handles the player's turn, prompting for color and number of tokens to remove.

    # Validates the input and updates the token counts.

    print(f"Player {player}'s turn")

    color = input("Choose a color (red/blue): ").strip().lower()

    while color not in ["red", "blue"]:

        color = input("Invalid color. Choose a color (red/blue): ").strip().lower()

    count = int(input("How many tokens to remove (1, 2, or 3)? "))

    while count not in [1, 2, 3]:

        count = int(input("Invalid number of tokens. Choose 1, 2, or 3: "))

    if color == "red":

        if red\_tokens >= count:

            red\_tokens -= count

        else:

            print(f"Not enough red tokens to remove {count}.")

            return player\_turn(player, red\_tokens, blue\_tokens)

    else:

        if blue\_tokens >= count:

            blue\_tokens -= count

        else:

            print(f"Not enough blue tokens to remove {count}.")

            return player\_turn(player, red\_tokens, blue\_tokens)

    return red\_tokens, blue\_tokens

def check\_winner\_standard(red\_tokens, blue\_tokens):

    # Determines if the game has ended in the standard version, where either pile is empty.

    return red\_tokens == 0 or blue\_tokens == 0

def check\_winner\_misere(red\_tokens, blue\_tokens):

    # Determines if the game has ended in the misère version, where either pile is empty.

    return red\_tokens == 0 or blue\_tokens == 0

def calculate\_score(red\_tokens, blue\_tokens):

    # Calculates the score based on the remaining tokens.

    # Each red token is worth 2 points, each blue token is worth 3 points.

    return red\_tokens \* 2 + blue\_tokens \* 3

def minimax(red\_tokens, blue\_tokens, depth, is\_maximizing, alpha, beta, version):

    # Implements the Minimax algorithm with Alpha-Beta Pruning to determine the optimal move.

    # Parameters:

    # - red\_tokens, blue\_tokens: Current state of tokens.

    # - depth: Remaining search depth.

    # - is\_maximizing: Boolean indicating if the current player is maximizing.

    # - alpha, beta: Alpha-Beta values for pruning.

    # - version: Game version ('standard' or 'misere').

    # Returns:

    # - Evaluated score for the current game state.

    if (version == 'standard' and check\_winner\_standard(red\_tokens, blue\_tokens)) or \

       (version == 'misere' and check\_winner\_misere(red\_tokens, blue\_tokens)) or depth == 0:

        return calculate\_score(red\_tokens, blue\_tokens)

    if is\_maximizing:

        max\_eval = float('-inf')

        for count in [1, 2, 3]:

            if red\_tokens >= count:

                eval = minimax(red\_tokens - count, blue\_tokens, depth - 1, False, alpha, beta, version)

                max\_eval = max(max\_eval, eval)

                alpha = max(alpha, eval)

                if beta <= alpha:

                    return max\_eval

            if blue\_tokens >= count:

                eval = minimax(red\_tokens, blue\_tokens - count, depth - 1, False, alpha, beta, version)

                max\_eval = max(max\_eval, eval)

                alpha = max(alpha, eval)

                if beta <= alpha:

                    return max\_eval

        return max\_eval

    else:

        min\_eval = float('inf')

        for count in [1, 2, 3]:

            if red\_tokens >= count:

                eval = minimax(red\_tokens - count, blue\_tokens, depth - 1, True, alpha, beta, version)

                min\_eval = min(min\_eval, eval)

                beta = min(beta, eval)

                if beta <= alpha:

                    return min\_eval

            if blue\_tokens >= count:

                eval = minimax(red\_tokens, blue\_tokens - count, depth - 1, True, alpha, beta, version)

                min\_eval = min(min\_eval, eval)

                beta = min(beta, eval)

                if beta <= alpha:

                    return min\_eval

        return min\_eval

def computer\_turn(red\_tokens, blue\_tokens, depth, version):

    # Determines the computer's optimal move using the Minimax algorithm.

    best\_score = float('-inf')

    best\_move = None

    for count in [1, 2, 3]:

        if red\_tokens >= count:

            score = minimax(red\_tokens - count, blue\_tokens, depth - 1, False, float('-inf'), float('inf'), version)

            if score > best\_score:

                best\_score = score

                best\_move = ('red', count)

        if blue\_tokens >= count:

            score = minimax(red\_tokens, blue\_tokens - count, depth - 1, False, float('-inf'), float('inf'), version)

            if score > best\_score:

                best\_score = score

                best\_move = ('blue', count)

    if best\_move is None:

        print("No valid moves available for computer.")

        return red\_tokens, blue\_tokens

    color, count = best\_move

    if color == 'red':

        red\_tokens -= count

    else:

        blue\_tokens -= count

    print(f"Computer removes {count} {color} tokens.")

    return red\_tokens, blue\_tokens

def red\_blue\_nim\_game():

    # Main function to start the game, handle user input, and control game flow.

    if len(sys.argv) < 4:

        print("Usage: python <name\_of\_the\_file> <num-red> <num-blue> <version> [<first-player> <depth>]")

        print("Examples:")

        print("  python <name\_of\_the\_file> 10 10 standard computer 3")

        print("  python <name\_of\_the\_file> 10 10 misere")

        print("  python <name\_of\_the\_file> 10 10 standard")

        return

    red\_tokens = int(sys.argv[1])

    blue\_tokens = int(sys.argv[2])

    version = sys.argv[3].lower()

    first\_player = sys.argv[4].lower() if len(sys.argv) > 4 else 'computer'

    depth = int(sys.argv[5]) if len(sys.argv) > 5 else 3

    current\_player = 1 if first\_player == 'human' else 2

    if version not in ['standard', 'misere']:

        print("Invalid game version. Choose 'standard' or 'misere'.")

        return

    while True:

        display\_tokens(red\_tokens, blue\_tokens)

        if current\_player == 1:

            red\_tokens, blue\_tokens = player\_turn("Human", red\_tokens, blue\_tokens)

        else:

            red\_tokens, blue\_tokens = computer\_turn(red\_tokens, blue\_tokens, depth, version)

        if (version == 'standard' and check\_winner\_standard(red\_tokens, blue\_tokens)) or \

           (version == 'misere' and check\_winner\_misere(red\_tokens, blue\_tokens)):

            print(f"Player {current\_player} wins!")

            break

        current\_player = 2 if current\_player == 1 else 1

    score = calculate\_score(red\_tokens, blue\_tokens)

    print(f"Final score: {score} points")

if \_\_name\_\_ == "\_\_main\_\_":

    red\_blue\_nim\_game()

**output screenshot:**

