MinimaxInfo is a struct or class that stores the minimax value of a state *and* an action representing the best move from that state.

table is a hash table or dictionary that stores a mapping between game states and MinimaxInfo objects.

Terminal-Test(state) is a function that returns true if state is a terminal state (meaning the game has been won, lost, or drawn).

Utility(state) is a function that determines the numerical worth of a *terminal* state. Typically these values are positive for a MAX win and negative for a MIN win, with 0 meaning a draw.

Cutoff-Test(state, depth) is a Boolean function that returns true if the search should be cut off at this state. Usually this is based on the depth of the state in the search tree; for instance, we could choose to cut off all searches after looking four moves ahead. This function can be based on other factors of the state, however, if desired, leading to some branches of the search tree searching deeper than other branches.

Eval(state) is a function that uses a heuristic to determine the numerical worth of a *non-terminal* state. Like the **Utility** function, these values should be positive for states that will likely lead to MAX wins, and negative for states that will likely lead to MIN wins. For MAX, higher values (closer to positive infinity) should correlate with a higher probability of winning. For MIN, lower values (closer to negative infinity) should correlate with a higher probability of winning.

Actions(state) is a function that returns all *legal* actions from a state.

Result(state, action) is a function that takes a state and an action and returns a new state (the successor state, or child state) that results from taking the action in the original state. This function assumes the action is a legal action from the state.

PlayerWhoMovesNext(state) is a function that takes a state and returns the player who moves next from that state. (This might just be a variable stored in the state object itself.)

Important Notes:

- This version of minimax with alpha-beta pruning and heuristics is designed to be re-run before every computer move, starting with an empty transposition table. The call should look like

  AlphaBetaWithHeuristics(current_state, table, negative infinity, positive infinity, 1)

  The transposition table must be reset for each call because each search will re-populate the table with states found deeper and deeper in the tree; this will likely result in different heuristic values bubbling up and may change the pruning process of alpha-beta.

  Every call to this function should start from the *current* state (the state from which the computer has to make a move). Do not re-search from the start state (the empty board).

- States that are short-circuited are not stored in the transposition table. They correspond to states that will never be chosen anyway, so there's no reason to store them. The code below reflects this. (If you do store such states in the table, nothing really changes; it just uses more memory.)

- Do not be concerned if your transposition table sizes do not match mine for Part B (they should match for Part A). The size of the transposition table is highly dependent on which moves are examined first by alpha-beta and the values returned by the heuristic function, and these two things may not match up exactly with my code (which is fine).

```
function AlphaBetaWithHeuristics(state, table, alpha, beta, depth):
    // if we've already computed the minimax value for this state, don't do it again!
    if table contains state:
        return table[state].minimaxValue

    else if Terminal-Test(state):
        u = Utility(state)
        table[state] = MinimaxInfo(u, null)        // terminal states have no best move
        return u

    else if Cutoff-Test(state, depth):
        e = Eval(state)
        table[state] = MinimaxInfo(e, null)        // search is ending early, so we don't know the best move
        return e

    else if PlayerWhoMovesNext(state) == MAX:
        bestMinimaxSoFar = negative infinity
        bestMoveForState = null
        for each action a in Actions(state):
            childState = Result(state, a)
            minimaxOfChild = AlphaBetaWithHeuristics(childState, table, alpha, beta, depth+1)
            if minimaxOfChild > bestMinimaxSoFar:
                bestMinimaxSoFar = minimaxOfChild
                bestMoveForState = a
            if minimaxSoFar >= beta:                        // short-circuit; prune search
                return bestMinimaxSoFar
            alpha = max(alpha, bestMinimaxSoFar)
        table[state] = MinimaxInfo(bestMinimaxSoFar, bestMoveForState)
        return bestMinimaxSoFar

    else // PlayerWhoMovesNext(state) == MIN:
        bestMinimaxSoFar = infinity
        bestMoveForState = null
        for each action a in Actions(state):
            childState = Result(state, a)
            minimaxOfChild = AlphaBetaWithHeuristics(childState, table, alpha, beta, depth+1)
            if minimaxOfChild < bestMinimaxSoFar:
                bestMinimaxSoFar = minimaxOfChild
                bestMoveForState = a
            if minimaxSoFar <= alpha:                       // short-circuit, prune search
                return bestMinimaxSoFar
            beta = min(beta, bestMinimaxSoFar)
        table[state] = MinimaxInfo(bestMinimaxSoFar, bestMoveForState)
        return bestMinimaxSoFar
```