2868. The Wording Game

Problem Description

Alice and Bob are playing a game using two lexicographically sorted lists of strings, `a` and `b`. The game is turn-based, starting with Alice. On a player's turn, they must choose a word from their list that is "closely greater" than the last word played. A word `w` is considered "closely greater" than a word `z` if `w` is lexicographically greater than `z`, and the first letter of `w` is either identical to or immediately follows the first letter of `z` in the alphabet. A player loses if they cannot play a word on their turn. The problem asks us to determine if Alice can win the game, assuming both players play optimally.

For instance, "car" can be followed by "care" or "cat" but not by "ant" or "book". If the previous word is "cook," the next valid word could be "cool" or "coop," but not "card," since "d" is not the letter immediately after "c".

The lexicographical comparison works in a way where the difference between two strings is found at the first differing character; the string with the higher alphabetical character at that position is considered greater. If there's no difference within the shorter length of the two strings, the longer string is deemed greater.

Intuition

The given solution approach is a simulation of the game, adhering to the rules described. The intuition behind this simulation is to track the current word and the indices for Alice and Bob's lists, marking whose turn it is with `k` (0 for Alice, 1 for Bob). Both players start with the smallest possible valid word from their respective lists.

The key observation here is to decide, for the current player, if their next word in the list is a valid "closely greater" word than the last word played. Because the lists are sorted lexicographically, the players can simply iterate through their lists in order, checking each word until they find a suitable one or run out of words.

Alice starts the game, so we begin with her list (except the first word, as it's already played). At each player's turn, we consider the following:

- If the player has no more words to play (reaching the end of their list), they lose.
- If the current word on the player's list is "closely greater," it is played, and the turn switches to the other player.
- If the current word is not valid, we move to the next word in the list.

The algorithm toggles between the players and iterates through both lists until it finds the outcome (either Alice can't play and loses, or Bob can't play and thus Alice wins). By following this optimal strategy simulation for both Alice and Bob, the solution is able to determine if Alice has a winning strategy.

Solution Approach

The implementation of the game simulation involves a while loop that continues indefinitely until a player cannot make a move. The solution relies on a few important observations about the nature of the game and the structure of the data:

1. Lexicographic Order: Since both lists `a` and `b` are sorted lexicographically, once a player cannot find a "closely greater" word after a certain word in their list, no word thereafter in their list can be a valid play. This is because each subsequent word is not only lexicographically greater but also differs by more than one letter from the beginning of the word, violating the "closely greater" condition.
2. Alternating Turns: The variable `k` indicates which player's turn it is. It starts with Alice (`k = 0`), and every time a valid move is made, `k` is toggled (`k ^= 1`). If `k` is 1 (Bob's turn), the simulation checks Bob's list; otherwise, it checks Alice's.
3. Pointer Iteration: Two pointers (`i` for Alice's index and `j` for Bob's index) iterate through their respective lists. These increase independently as each player takes their turn, meaning they only look for a "closely greater" word from where they left off last time.
4. Valid Move Check: On a player's turn, for their current word to be a valid "closely greater" move, one of two conditions must be met:
   - The first letters of the current word and the last played word are the same, and the current word is lexicographically greater.
   - The first letter of the current word is exactly one letter after the first letter of the last played word in the alphabet.

This is implemented using conditional statements checking `b[j][0] == w[0] and b[j] > w or ord(b[j][0]) - ord(w[0]) == 1` for Bob, and similarly for Alice.

Once the simulation finds that a player cannot play any more words (the pointer has reached the end of their list), it exits the loop. This is programmed as `if j == len(b)` for Bob (Alice wins), and `if i == len(a)` for Alice (Bob wins).

This results in a neat and efficient simulation of the game, effectively modeling each player's possible moves without having to backtrack or reconsider previous decisions. The efficiency stems from both the sorted nature of the lists and the binary nature of the turn-taking, which allows the solution to straightforwardly execute the optimal strategy simulation for both players.

Example Walkthrough

Let's walk through a small example using the solution approach described.

Assume the sorted lists for Alice and Bob are as follows:

Alice's list `a`: `["arc", "bit", "cart"]` Bob's list `b`: `["art", "car", "cute", "dart"]`

Now, let's simulate the game:

1. Alice starts (`k = 0`) and plays the smallest word from her list, which is "arc".
2. It's now Bob's turn (`k = 1`). The first word in his list, "art", is "closely greater" to "arc" (same start letter and lexicographically greater), so he plays "art".
3. Alice can now no longer play "arc" or "bit" because they are not "closely greater" than "art". She plays her next valid word "cart".
4. Bob's next word is "car", but "car" isn't valid because it's not greater than "cart". He skips to "cute", which also isn't valid (the first letter 'c' doesn't follow 'c' directly in the alphabet), so he has to play "dart".
5. Alice can't play "arc" or "bit" as they were already invalid. Her last word is "cart", which is not "closely greater" than "dart", and there are no remaining words in her list.
6. Alice fails to play a move, so Bob wins.

In this walkthrough, Bob has won the game by playing his words efficiently, and Alice reaches the end of her list without a valid move to make. This example demonstrates the solution approach where each player, on their turn, chooses the smallest valid "closely greater" word from their list, until one player ultimately cannot make a move and loses.

Solution Implementation

Python

```python
class Solution:
    def canAliceWin(self, alice_cards: List[str], bob_cards: List[str]) -> bool:
        # Initialize pointers for Alice and Bob's card and a flag for Alice's turn.
        alice_index, bob_index = 1, 0
        is_alice_turn = True

        # Winning card starts as Alice's first card.
        winning_card = alice_cards[0]

        # Loop until break condition is met.
        while True:
            # Check if it's Alice's turn.
            if is_alice_turn:
                # If Bob has no more cards, Alice wins.
                if bob_index == len(bob_cards):
                    return True
                # Compare cards and update the winning card if conditions are met.
                if (bob_cards[bob_index][0] == winning_card[0] and bob_cards[bob_index] > winning_card) \
                or ord(bob_cards[bob_index][0]) - ord(winning_card[0]) == 1:
                    winning_card = bob_cards[bob_index]
                    # Toggle the turn.
                    is_alice_turn = not is_alice_turn
                # Move to the next card for Bob.
                bob_index += 1
            else:
                # If Alice has no more cards, Bob wins, hence Alice loses.
                if alice_index == len(alice_cards):
                    return False
                # Compare cards and update the winning card if conditions are met.
                if (alice_cards[alice_index][0] == winning_card[0] and alice_cards[alice_index] > winning_card) \
                or ord(alice_cards[alice_index][0]) - ord(winning_card[0]) == 1:
                    winning_card = alice_cards[alice_index]
                    # Toggle the turn.
                    is_alice_turn = not is_alice_turn
                # Move to the next card for Alice.
                alice_index += 1
```

Java

```java
class Solution {
    public boolean canAliceWin(String[] aliceCards, String[] bobCards) {
        // Initialize pointer i for Alice's cards and j for Bob's cards
        int aliceIndex = 1, bobIndex = 0;

        // Boolean flag to keep track of whose turn it is; true for Alice's turn and false for Bob's turn
        boolean isAliceTurn = true;

        // The card to compare with, beginning with the first card of Alice's
        String currentWinningCard = aliceCards[0];

        // Infinite loop to simulate the turns taken by Alice and Bob
        while (true) {
            // If it's Alice's turn
            if (isAliceTurn) {
                // If all Bob's cards are played, Alice wins
                if (bobIndex == bobCards.length) {
                    return true;
                }

                // Alice can play a card only if it's directly greater than the current winning card,
                // or if it's one rank higher regardless of the suit. If played, swap turns
                if ((bobCards[bobIndex].charAt(0) == currentWinningCard.charAt(0) && currentWinningCard.compareTo(bobCards[bobIndex]) < 0)
                    || (bobCards[bobIndex].charAt(0) - currentWinningCard.charAt(0) == 1)) {
                    currentWinningCard = bobCards[bobIndex];
                    isAliceTurn = !isAliceTurn;
                }
                // Move to the next card in Bob's deck
                ++bobIndex;
            } else {
                // If all Alice's cards are played, Bob wins, hence Alice loses
                if (aliceIndex == aliceCards.length) {
                    return false;
                }

                // Bob can play a card only if it's directly greater than the current winning card,
                // or if it's one rank higher regardless of the suit. If played, swap turns
                if ((aliceCards[aliceIndex].charAt(0) == currentWinningCard.charAt(0) && currentWinningCard.compareTo(aliceCards[aliceIndex]) < 0)
                    || (aliceCards[aliceIndex].charAt(0) - currentWinningCard.charAt(0) == 1)) {
                    currentWinningCard = aliceCards[aliceIndex];
                    isAliceTurn = !isAliceTurn;
                }
                // Move to the next card in Alice's deck
                ++aliceIndex;
            }
        }
    }
}
```

C++

```cpp
class Solution {
public:
    bool canAliceWin(vector<string>& aliceWords, vector<string>& bobWords) {
        int aliceIndex = 1; // Start with the second word for Alice as she starts the game with the first word
        int bobIndex = 0;   // Start with the first word for Bob
        bool isAliceTurn = true; // Alice starts the game, so it's her turn initially
        string currentWord = aliceWords[0]; // Set the current word as Alice's first word

        // Continue till a decision can be reached
        while (true) {
            if (isAliceTurn) { // If it's Alice's turn
                if (bobIndex == bobWords.size()) { // If Bob has no more words to play, Alice wins
                    return true;
                }
                // Check if the first letter of Bob's current word follows the current word alphabetically or is the same starting le
                if ((bobWords[bobIndex][0] == currentWord[0] && currentWord < bobWords[bobIndex]) || bobWords[bobIndex][0] - currentW
                    currentWord = bobWords[bobIndex]; // Update the current word
                    isAliceTurn ^= 1; // Toggle the turn to Bob's
                }
                bobIndex++; // Move to Bob's next word
            } else { // If it's Bob's turn
                if (aliceIndex == aliceWords.size()) { // If Alice has no more words to play, Bob wins, so Alice loses
                    return false;
                }
                // Check if the first letter of Alice's current word follows the current word alphabetically or is the same starting
                if (aliceWords[aliceIndex][0] == currentWord[0] && currentWord < aliceWords[aliceIndex] || aliceWords[aliceIndex][0
                    currentWord = aliceWords[aliceIndex]; // Update the current word
                    isAliceTurn ^= 1; // Toggle the turn to Alice's
                }
                aliceIndex++; // Move to Alice's next word
            }
        }
    }
};
```

TypeScript

```typescript
function canAliceWin(aliceCards: string[], bobCards: string[]): boolean {
    // Initialize indices for Alice (i) and Bob (ii), and a flag (turnFlag) to track whose turn it is.
    // Alice starts with the first card in her array (index 0).
    let aliceIndex = 1;
    let bobIndex = 0;
    let turnFlag = 1; // 1 for Alice's turn, 0 for Bob's turn.
    let currentWinningCard = aliceCards[0]; // Current winning card, starting with Alice's first card.

    // Continue the game until a winner is determined.
    while (true) {
        if (turnFlag === 1) { // If it's Alice's turn
            if (bobIndex === bobCards.length) {
                return true; // Alice wins if Bob has no more cards.
            }

            // Check if Bob has a card that beats the current winning card, either by suit or rank.
            if ((bobCards[bobIndex][0] === currentWinningCard[0] && currentWinningCard < bobCards[bobIndex]) ||
                bobCards[bobIndex].charCodeAt(0) - currentWinningCard.charCodeAt(0) === 1) {
                currentWinningCard = bobCards[bobIndex]; // Bob takes the lead with a new winning card.
                turnFlag ^= 1; // Toggle the turn flag to switch turns.
            }

            bobIndex++; // Move to Bob's next card.
        } else { // If it's Bob's turn
            if (aliceIndex === aliceCards.length) {
                return false; // Bob wins if Alice has no more cards.
            }

            // Check if Alice has a card that beats the current winning card, either by suit or rank.
            if ((aliceCards[aliceIndex][0] === currentWinningCard[0] && currentWinningCard < aliceCards[aliceIndex]) ||
                aliceCards[aliceIndex].charCodeAt(0) - currentWinningCard.charCodeAt(0) === 1) {
                currentWinningCard = aliceCards[aliceIndex]; // Alice takes the lead with a new winning card.
                turnFlag ^= 1; // Toggle the turn flag to switch turns.
            }

            aliceIndex++; // Move to Alice's next card.
        }
    }
}
```

```python
class Solution:
    def canAliceWin(self, alice_cards: List[str], bob_cards: List[str]) -> bool:
        # Initialize pointers for Alice and Bob's card and a flag for Alice's turn.
        alice_index, bob_index = 1, 0
        is_alice_turn = True

        # Winning card starts as Alice's first card.
        winning_card = alice_cards[0]

        # Loop until break condition is met.
        while True:
            # Check if it's Alice's turn.
            if is_alice_turn:
                # If Bob has no more cards, Alice wins.
                if bob_index == len(bob_cards):
                    return True
                # Compare cards and update the winning card if conditions are met.
                if (bob_cards[bob_index][0] == winning_card[0] and bob_cards[bob_index] > winning_card) \
                or ord(bob_cards[bob_index][0]) - ord(winning_card[0]) == 1:
                    winning_card = bob_cards[bob_index]
                    # Toggle the turn.
                    is_alice_turn = not is_alice_turn
                # Move to the next card for Bob.
                bob_index += 1
            else:
                # If Alice has no more cards, Bob wins, hence Alice loses.
                if alice_index == len(alice_cards):
                    return False
                # Compare cards and update the winning card if conditions are met.
                if (alice_cards[alice_index][0] == winning_card[0] and alice_cards[alice_index] > winning_card) \
                or ord(alice_cards[alice_index][0]) - ord(winning_card[0]) == 1:
                    winning_card = alice_cards[alice_index]
                    # Toggle the turn.
                    is_alice_turn = not is_alice_turn
                # Move to the next card for Alice.
                alice_index += 1
```

Time and Space Complexity

The time complexity of the provided code is $O(m + n)$, where m is the length of list `a` and n is the length of list `b`. This is because the code involves a single while loop with no nested loops, and it traverses each list at most once, incrementally iterating the indices `i` and `j`.

The space complexity of the code is $O(1)$. This is because the amount of additional memory used does not depend on the size of the input lists `a` and `b`, but is rather constant. Only a fixed amount of space for the variables `i`, `j`, `k`, and `w` is needed, regardless of input size.