1686. Stone Game VI

Medium Greedy Array Math Game Theory Sorting Heap (Priority Queue) Leetcode Link

Problem Description

Alice and Bob are playing a game with `n` stones arranged in a pile. Each stone has a value associated with it, which means how much it is worth to Alice and to Bob. For each player, the worth of each stone is known and represented by two integer arrays `aliceValues` and `bobValues`. Every turn, a player can remove a stone from the pile and score the amount that stone is worth to them.

The goal of the game is to end up with more points than the opponent by the time all stones are removed from the pile. If Alice and Bob end up with the same score, the game is considered a draw. Both Alice and Bob play optimally, meaning they each aim to finish the game with the maximum possible points for themselves, knowing fully well the strategy and values of the other player.

The task is to determine the result of the game:

- If Alice wins, the function should return `1`.
- If Bob wins, the function should return `-1`.
- If the game is a draw, the function should return `0`.

Intuition

To achieve the solution for this game, we must think about how each player would play optimally. The key is to realize that at any turn, the player will choose a stone not only based on how much it's worth to them but also considering how valuable it is to their opponent. By doing so, they can maximize their advantage or minimize the opponent's score in the subsequent turn.

The intuition behind the solution is to consider the total value of each stone, which is the sum of its value to both Alice and Bob. By creating a list of tuples, where each tuple consists of the total value of a stone and its index, we can sort this list in descending order. This way, the first element of this sorted list gives us the stone that has the highest impact on the game when removed.

Once we have this sorted list, we can simulate the game by iterating over this list and adding the value of the stone to Alice's score if it is her turn (every even index) and to Bob's score if it's his turn (every odd index). Since the list is sorted by the total impact on the game, we ensure that each player picks the optimal stone at their turn.

After distributing all the stones and calculating the total scores for Alice and Bob, we compare their scores:

- If Alice's score is higher than Bob's, we return `1`.
- If Bob's score is higher than Alice's, we return `-1`.
- If both scores are equal, it results in a draw, and we return `0`.

Solution Approach

The implementation uses a greedy strategy to play the game optimally. By looking at both the `aliceValues` and `bobValues`, we create a list that combines these values into a single measurement of a stone's impact on the game. This is done using the following step:

1. Pair each stone's `aliceValues[i]` and `bobValues[i]` together and calculate their sum. We create an array of tuples (`arr`) where each tuple holds the total value of a stone (`a + b`) and its index (`i`).

The created list is then sorted in descending order based on the total value of each stone. This is performed using Python's `sort()` method with `reverse=True`.

2. With the sorted list `arr`, we iterate over it, alternating turns between Alice and Bob (Alice starting first, as per the rules of the game). We use a list comprehension to sum the `aliceValues` for the stones chosen by Alice (indexed at even positions in the sorted list), and similarly for Bob (indexed at odd positions). This is handled by these two lines of code:

```python
a = sum(aliceValues[v[1]] for i, v in enumerate(arr) if i % 2 == 0)
b = sum(bobValues[v[1]] for i, v in enumerate(arr) if i % 2 == 1)
```

Here, `a` represents Alice's total score, and `b` is Bob's total score.

3. Finally, we determine the winner by comparing Alice's and Bob's scores:
   - If `a` > `b`, Alice has more points and thus we return `1`.
   - If `a` < `b`, Bob has more points and we return `-1`.
   - If they are equal (`a == b`), it's a draw, and we return `0`.

This approach effectively simulates the best possible strategy for both players, taking turns in a greedy manner to either maximize their own score or prevent the opponent from gaining a significant scoring stone. Because both players play optimally, it is implied that the obtained result reflects the outcome of an actual game played between two optimally playing individuals given the same set of values.

Example Walkthrough

Consider a small example with `n = 4` stones. The values of the stones to Alice `aliceValues` are `[5, 4, 3, 2]` and to Bob `bobValues` are `[4, 1, 2, 3]`.

Following the solution approach:

1. We first create an array of tuples representing the total value of each stone and its index:

```python
aliceValues = [5, 4, 3, 2]
bobValues = [4, 1, 2, 3]
arr = [(aliceValues[i] + bobValues[i], i) for i in range(n)]
# arr = [(9, 0), (5, 1), (5, 2), (5, 3)]
```

2. We sort `arr` in descending order by the total value:

```python
arr.sort(key=lambda x: x[0], reverse=True)
# arr = [(9, 0), (5, 1), (5, 2), (5, 3)] (Note: Already sorted in this case)
```

3. Alice and Bob take turns picking values based on their respective index positions in the `arr`. Alice starts first:

- Alice picks index `0` (highest total value). Her score `a` is now `5` (from `aliceValues`).
- Bob picks index `1`. His score `b` is now `1` (from `bobValues`).
- Alice picks index `2`. Her score `a` is now `5 + 3 = 8`.
- Bob picks index `3`. His score `b` is now `1 + 3 = 4`.

Thus, after this process, Alice's total score `a` is `8`, and Bob's total score `b` is `4`.

4. We compare Alice's score to Bob's:

- Since `a` (8) > `b` (4), Alice has won the game, and we return `1`.

Using this solution approach on the given example, we can see that Alice would indeed win the game if both players play optimally. This demonstrates the greedy strategy used to decide which stone to pick each turn.

Python Solution

```python
from typing import List

class Solution:
    def stoneGameVI(self, alice_values: List[int], bob_values: List[int]) -> int:
        # Combine the values for Alice and Bob into a single list, tracking the index.
        combined_values = [(alice + bob, index) for index, (alice, bob) in enumerate(zip(alice_values, bob_values))]

        # Sort the combined values in descending order to prioritize the stones with the higher sums.
        combined_values.sort(reverse=True)

        # Calculate Alice's score by summing up the values at the even indices.
        alice_score = sum(alice_values[index] for i, (_, index) in enumerate(combined_values) if i % 2 == 0)

        # Calculate Bob's score by summing up the values at the odd indices.
        bob_score = sum(bob_values[index] for i, (_, index) in enumerate(combined_values) if i % 2 == 1)

        # Determine the result based on the comparison of Alice's and Bob's scores.
        if alice_score > bob_score:
            return 1  # Alice wins
        if alice_score < bob_score:
            return -1  # Bob wins
        return 0  # Tie

# The method stoneGameVI takes two lists of integers: 'alice_values' and 'bob_values'.
# Each list contains the values of the stones from Alice's and Bob's perspective, respectively.
# The method returns 1 if Alice can get more points, -1 if Bob can get more points, or 0 for a tie.

```

Java Solution

```java
class Solution {

    public int stoneGameVI(int[] aliceValues, int[] bobValues) {
        // Length of the arrays representing the stones' values.
        int n = aliceValues.length;

        // Initialize a 2D array to keep pairs of sums of values (Alice + Bob) and their indices.
        int[][] valuePairs = new int[n][2];

        for (int i = 0; i < n; ++i) {
            // Sum the values of Alice's and Bob's stones and store with the index.
            valuePairs[i] = new int[] {aliceValues[i] + bobValues[i], i};
        }

        // Sort the array based on the sum of the values in descending order.
        Arrays.sort(valuePairs, (a, b) -> b[0] - a[0]);

        // Initialize the scores of Alice and Bob as 0.
        int aliceScore = 0, bobScore = 0;

        for (int i = 0; i < n; ++i) {
            // Retrieve the original index of the stone.
            int index = valuePairs[i][1];

            // Alice picks first, then Bob picks, and continue in this manner.
            if (i % 2 == 0) {
                aliceScore += aliceValues[index]; // Alice's turn (even turns)
            } else {
                bobScore += bobValues[index]; // Bob's turn (odd turns)
            }
        }

        // Compare scores and return the result according to the problem's convention.
        if (aliceScore == bobScore) {
            return 0; // No one has a higher score.
        }
        return aliceScore > bobScore ? 1 : -1; // Alice (1) or Bob (-1) has a higher score.
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    // Method to determine the winner of the stone game.
    // Return 0 if tied, 1 if Alice wins, -1 if Bob wins.
    int stoneGameVI(vector<int>& aliceValues, vector<int>& bobValues) {
        int n = aliceValues.size(); // Number of stones

        // Creating a vector to store the sum of Alice's and Bob's values, paired with the index
        vector<pair<int, int>> valuePairs(n);
        for (int i = 0; i < n; ++i) {
            valuePairs[i] = {aliceValues[i] + bobValues[i], i}; // Pair sum of values with the index
        }

        // Sort the pairs in descending order based on the sum of values
        sort(valuePairs.rbegin(), valuePairs.rend());

        // Initialize scores for Alice and Bob
        int aliceScore = 0, bobScore = 0;

        // Picking stones in the order of decreasing sum of values
        for (int i = 0; i < n; ++i) {
            int index = valuePairs[i].second; // Retrieve the index of the original value
            // If i is even, it is Alice's turn, else it's Bob's turn
            if (i % 2 == 0) {
                aliceScore += aliceValues[index]; // Alice picks the stone
            } else {
                bobScore += bobValues[index]; // Bob picks the stone
            }
        }

        // Compare final scores to decide the winner
        if (aliceScore == bobScore) return 0; // Game is tied
        return aliceScore > bobScore ? 1 : -1; // If Alice's score is higher she wins, otherwise Bob wins
    }
};

```

Typescript Solution

```typescript
// Define the structure for pairing values with indices
interface ValuePair {
    sum: number;
    index: number;
}

// Function to determine the winner of the stone game.
// Returns 0 if tied, 1 if Alice wins, -1 if Bob wins.
function stoneGameVI(aliceValues: number[], bobValues: number[]): number {
    const n: number = aliceValues.length; // Number of stones

    // Create an array to store the sum of values from both Alice and Bob, along with the index
    let valuePairs: ValuePair[] = [];
    for (let i = 0; i < n; ++i) {
        valuePairs.push({sum: aliceValues[i] + bobValues[i], index: i});
    }

    // Sort the pairs in descending order based on the sum of values
    valuePairs.sort((a, b) => b.sum - a.sum);

    // Initialize scores for Alice and Bob
    let aliceScore: number = 0, bobScore: number = 0;

    // Iterating over stones in the order of decreasing sum of values
    for (let i = 0; i < n; ++i) {
        const index: number = valuePairs[i].index; // Retrieve the original index
        // If the index of the array is even, it's Alice's turn; otherwise, it's Bob's turn
        if (i % 2 === 0) {
            aliceScore += aliceValues[index]; // Alice picks the stone
        } else {
            bobScore += bobValues[index]; // Bob picks the stone
        }
    }

    // Compare final scores to decide the winner
    if (aliceScore === bobScore) return 0; // Game is tied
    return aliceScore > bobScore ? 1 : -1; // If Alice's score is higher, she wins; otherwise, Bob wins
}

```

Time and Space Complexity

Time Complexity

The time complexity of the function `stoneGameVI` is determined by several operations:

- **Zip and Enumerate:** Both operations, `zip(aliceValues, bobValues)` and `enumerate`, are `O(n)` where `n` is the length of the input lists `aliceValues` and `bobValues`.
- **List Comprehension:** Constructing the `arr` list consisting of tuples `(a + b, i)` is also `O(n)` because it processes each element of the zipped lists once.
- **Sorting:** The sort operation `arr.sort(reverse=True)` is the most computationally expensive, with a time complexity of `O(n log n)` as TimSort (Python's sorting algorithm) is used.
- **Summation with Conditional Logic:** Two separate summations are involved, both of which iterate over the sorted `arr`. The iterations themselves are `O(n)`. Since we perform this action twice, this doesn't change the overall linear `O(n)` factor since they are not nested loops.

Combining them, the dominant factor and therefore the overall time complexity is `O(n log n)` because the `sort` operation grows faster than the linear steps as the input size increases.

Space Complexity

For space complexity:

- **List `arr`:** The list `arr` is of size `n`, thus requires `O(n)` additional space.
- **Variables `a` and `b`:** Since `a` and `b` are just accumulators, they require `O(1)` space.
- **Temporary Variables During Sorting:** `sort()` functions typically use `O(n)` space because of the way they manage intermediate values during sorting.

Hence, the overall space complexity of `stoneGameVI` is `O(n)`, accounting for the additional list and the space required for sorting.