1686. Stone Game VI

Array Math

Game Theory

# **Problem Description**

Medium

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.

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

Sorting Heap (Priority Queue)

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 the game is a draw, the function should return 0.

• If Bob wins, the function should return -1.

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

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.

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

### 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).

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:

b = sum(bobValues[v[1]] for i, v in enumerate(arr) if i % 2 == 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

```
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.
```

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

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

# 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).

```
optimally. This demonstrates the greedy strategy used to decide which stone to pick each turn.
```

# Combine the values for Alice and Bob into a single list, tracking the index.

# Calculate Bob's score by summing up the values at the odd indices.

# Determine the result based on the comparison of Alice's and Bob's scores.

# 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.

combined\_values = [(alice + bob, index) for index, (alice, bob) in enumerate(zip(alice\_values, bob\_values))]

alice\_score = sum(alice\_values[index] for i, (\_, index) in enumerate(combined\_values) if i % 2 == 0)

bob\_score = sum(bob\_values[index] for i, (\_, index) in enumerate(combined\_values) if i % 2 == 1)

// Initialize a 2D array to keep pairs of sums of values (Alice + Bob) and their indices.

// 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.

Using this solution approach on the given example, we can see that Alice would indeed win the game if both players play

```
# 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.
```

return 0 # Tie

if alice score > bob score:

if alice score < bob score:</pre>

return -1 # Bob wins

int[][] valuePairs = new int[n][2];

int aliceScore = 0, bobScore = 0;

int index = valuePairs[i][1];

for (int i = 0; i < n; ++i) {

for (int i = 0; i < n; ++i) {

sort(valuePairs.rbegin(), valuePairs.rend());

// Compare final scores to decide the winner

// Function to determine the winner of the stone game.

// Returns 0 if tied, 1 if Alice wins, -1 if Bob wins.

let valuePairs: ValuePair[] = [];

valuePairs.sort((a, b) => b.sum - a.sum);

// Initialize scores for Alice and Bob

for (let i = 0; i < n; ++i) {

if (aliceScore == bobScore) return 0; // Game is tied

function stoneGameVI(aliceValues: number[], bobValues: number[]): number {

// Sort the pairs in descending order based on the sum of values

valuePairs.push({sum: aliceValues[i] + bobValues[i], index: i});

const n: number = aliceValues.length; // Number of stones

// Initialize scores for Alice and Bob

int aliceScore = 0, bobScore = 0;

for (int i = 0; i < n; ++i) {

if (i % 2 == 0) {

} else {

Arrays.sort(valuePairs,  $(a, b) \rightarrow b[0] - a[0]$ );

// Initialize the scores of Alice and Bob as 0.

// Retrieve the original index of the stone.

for (int i = 0; i < n; ++i) {

return 1 # Alice wins

```
// 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.
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);
```

valuePairs[i] = {aliceValues[i] + bobValues[i], i}; // Pair sum of values with the index

return aliceScore > bobScore ? 1 : -1; // If Alice's score is higher she wins, otherwise Bob wins

int index = valuePairs[i].second; // Retrieve the index of the original value

// Create an array to store the sum of values from both Alice and Bob, along with the index

// Sort the pairs in descending order based on the sum of values

// If i is even, it is Alice's turn, else it's Bob's turn

bobScore += bobValues[index]; // Bob picks the stone

aliceScore += aliceValues[index]; // Alice picks the stone

// Picking stones in the order of decreasing sum of values

```
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
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:
```

# Time and Space Complexity **Time Complexity**

aliceValues and bobValues.

return 1 # Alice wins

return −1 # Bob wins

if alice score < bob score:</pre>

return 0 # Tie

## • List Comprehension: Constructing the arr list consisting of tuples (a + b, i) is also 0(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.

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

• Summation with Conditional Logic: Two separate summations are involved, both of which iterate over the sorted arr. The iterations

• Zip and Enumerate: Both operations, zip(aliceValues, bobValues) and enumerate, are O(n) where n is the length of the input lists

- 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 0(n log n) because the sort operation grows
- faster than the linear steps as the input size increases.

# **Space Complexity** For space complexity:

- 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 0(n), accounting for the additional list and the space required for sorting.

- Intuition
- 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.
- 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
- following step:
- 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.
- a = sum(aliceValues[v[1]] for i, v in enumerate(arr) if i % 2 == 0) 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.
- maximize their own score or prevent the opponent from gaining a significant scoring stone. Because both players play optimally, **Example Walkthrough**

bobValues are [4, 1, 2, 3].

Following the solution approach:

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

Solution Implementation

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

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

arr.sort(key=lambda x: x[0], reverse=True)

- 1. We first create an array of tuples representing the total value of each stone and its index: aliceValues = [5, 4, 3, 2]bobValues = [4, 1, 2, 3]
- 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.
- **Python** from typing import List class Solution: def stoneGameVI(self, alice values: List[int], bob values: List[int]) -> int:
- # The method stoneGameVI takes two lists of integers: 'alice values' and 'bob values'. Java class Solution { public int stoneGameVI(int[] aliceValues, int[] bobValues) { // Length of the arrays representing the stones' values. int n = aliceValues.length;

- **TypeScript** // Define the structure for pairing values with indices interface ValuePair {

sum: number:

index: number;

**}**;

- # 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.

  - 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 0(1) space.