735. Asteroid Collision

Medium Stack Array Simulation

Leetcode Link

Problem Description

In this problem, we are dealing with a simulation of asteroids moving in a row, represented by an array called `asteroids`. Each integer in the array represents an asteroid's size (absolute value) and direction (sign of the integer). Positive numbers mean the asteroid is moving to the right, while negative numbers mean the asteroid is moving to the left. All asteroids move at the same constant speed.

The task is to simulate the collisions that may happen when asteroids meet as they move. When two asteroids collide, the following rules apply:

- If one asteroid is larger than the other, the smaller asteroid is destroyed.
- If both asteroids are the same size, they both are destroyed.
- Asteroids moving in the same direction do not collide.

The goal is to determine the final state of the asteroids after all possible collisions have occurred.

Intuition

To solve this problem, we use a stack data structure that will help us manage the asteroid collisions efficiently. The stack is chosen because collisions affect the asteroids in a last-in, first-out manner: the most recently moving right asteroid can collide with the newly encountered left-moving one.

Here's the intuition for the solution approach:

- If we encounter an asteroid moving to the right (`x > 0`), it won't collide with any of the previous asteroids. We add (push) this asteroid to the stack, as it could collide with a future left-moving asteroid.
- When an asteroid moving to the left (`x < 0`) is found, we need to check if it will collide with any of the right-moving asteroids already in the stack.
  - If a collision is possible (the stack is not empty and the top asteroid in the stack is a positive number, meaning it is moving right), and the top asteroid in the stack is smaller than the current one (absolute size comparison), the top asteroid will explode (be removed by `pop` from the stack).
  - If the top asteroid in the stack has the same size (but opposite direction) as the current one, they both explode (we `pop` the top of the stack without adding the current one).
  - If the stack is empty or contains only asteroids moving to the left, it means there will be no collision for the current asteroid; therefore, we can safely add (push) the current asteroid to the stack.

By applying these steps, we simulate all possible collisions. Once we finish iterating through the `asteroids` array, the remaining asteroids in the stack will be the ones surviving after all collisions, which becomes our final result.

Solution Approach

The solution approach leverages a stack to keep track of the asteroids that are still in play, i.e., those that haven't been destroyed in a collision.

Here is the step-by-step approach based on the Reference Solution provided:

1. We initialize an empty list `stk` to act as our stack.
2. We iterate through each asteroid `x` in the `asteroids` array from left to right.
3. For an asteroid moving to the right (`x > 0`), we push it onto the stack since it cannot collide with asteroids already on the stack (as they are either also moving to the right or are larger left-moving asteroids that already survived previous collisions).
4. For an asteroid moving to the left (`x < 0`), we check for possible collisions with right-moving asteroids that are on top of the stack.
   - We continue to check the top of the stack (`stk[-1]`) to ensure the top asteroid is moving right (`stk[-1] > 0`) and is smaller than the current asteroid (`stk[-1] < -x`).
   - If both conditions are met, the top asteroid will explode; we remove it by popping it from the stack.
   - This pop operation is repeated in a loop until the stack's top asteroid is too large to be destroyed (`stk[-1] >= -x`) or until we find a left-moving asteroid (`stk[-1] < 0`), indicating that the current asteroid will not collide with any other asteroids in the stack.
5. If at any point, the asteroid on top of the stack is equal in size to the current one, both asteroids explode. Therefore, we pop the top asteroid from the stack without pushing the current one.
6. If, after checking for collisions, the stack is empty or the top asteroid is moving to the left, no collision occurs, and we push the current asteroid into the stack.
7. After processing all the asteroids in the given array, the remaining asteroids in the stack are those that survived all collisions. We return the `stk` as the final state of asteroids.

By using this approach, we effectively simulate asteroid movement and collisions, leading us to the final answer.

Example Walkthrough

Let's apply the solution approach on an example input of asteroids: `asteroids = [5, 10, -5]`.

We initialize an empty list `stk` to act as our stack. `stk = []`.

1. First, we encounter asteroid `5`, which is moving to the right (`x > 0`). We push it onto the stack. Now `stk = [5]`.
2. Next, we encounter asteroid `10`, which is also moving to the right (`x > 0`). We push it onto the stack as well. Now `stk = [5, 10]`.
3. Finally, we encounter asteroid `-5`, which is moving to the left (`x < 0`). We now check for possible collisions with asteroids on top of the stack:
   - The top of the stack is `10` which is moving to the right (`stk[-1] > 0`), but it is larger than our current asteroid (`stk[-1] = 10 > |-5| = 5`). Therefore, there's no pop operation, as `-5` is destroyed in the collision.
4. We continue and see that there are no more asteroids in the `asteroids` list and conclude processing.

The remaining asteroids in `stk` give us the final state after all possible collisions have occurred. In this case, the surviving asteroids are `[5,10]`. Therefore, our function would return this list.

This example illustrates how the stack is utilized to process the asteroids one by one, simulating the collisions based on their direction and size and leaving us with the final state of the asteroids.

Python Solution

```python
class Solution:
    def asteroidCollision(self, asteroids: List[int]) -> List[int]:
        # Initialize a stack to keep track of asteroids that are still moving
        stack = []

        # Iterate through each asteroid in the list
        for asteroid in asteroids:
            # If asteroid is moving to the right (positive), push it to the stack
            if asteroid > 0:
                stack.append(asteroid)
            else:
                # While there is at least one asteroid in the stack moving to the right
                # and the current left-moving asteroid is larger (in absolute value)
                # than the top asteroid in the stack, pop the top of the stack
                while stack and stack[-1] > 0 and stack[-1] < -asteroid:
                    stack.pop()

                # If the top of the stack is an asteroid with the same size as the
                # current one (but moving in the opposite direction), they both explode.
                if stack and stack[-1] == -asteroid:
                    stack.pop()

                # If the stack is empty or the top asteroid is moving to the left,
                # push the current asteroid to the stack
                elif not stack or stack[-1] < 0:
                    stack.append(asteroid)

        # Return the stack representing asteroids that survived the collisions
        return stack
```

Java Solution

```java
class Solution {
    public int[] asteroidCollision(int[] asteroids) {
        // Initialize a stack to keep track of the asteroids
        Deque<Integer> stack = new ArrayDeque<>();

        // Iterate through the array of asteroids
        for (int asteroid : asteroids) {
            // If the asteroid is moving to the right, push it onto the stack
            if (asteroid > 0) {
                stack.offerLast(asteroid);
            } else {
                // While there are asteroids in the stack moving to the right, and the
                // current asteroid's magnitude is larger, pop the asteroids from the stack
                while (!stack.isEmpty() && stack.peekLast() > 0 && stack.peekLast() < -asteroid) {
                    stack.pollLast();
                }

                // If the top of the stack is an asteroid of the same magnitude but moving in the opposite direction, destroy both
                if (!stack.isEmpty() && stack.peekLast() == -asteroid) {
                    stack.pollLast();
                } else if (stack.isEmpty() || stack.peekLast() < 0) {
                    // If there are no asteroids on the stack or the top asteroid is moving to the left, push the current asteroid or
                    stack.offerLast(asteroid);
                }
            }
        }

        // Convert the remaining asteroids in the stack to an array
        return stack.stream().mapToInt(Integer::valueOf).toArray();
    }
}
```

C++ Solution

```cpp
class Solution {
public:
    vector<int> asteroidCollision(vector<int>& asteroids) {
        vector<int> stack;
        // Loop through all asteroids
        for (int asteroid : asteroids) {
            // If asteroid is moving to the right (positive), add it to the stack.
            if (asteroid > 0) {
                stack.push_back(asteroid);
            } else {
                // While there are asteroids in the stack moving to the right (positive)
                // and the current asteroid is larger (negative magnitude comparison)
                while (!stack.empty() && stack.back() > 0 && stack.back() < -asteroid) {
                    stack.pop_back(); // Pop the smaller asteroids as they are destroyed
                }
                // If the top asteroid on the stack is the same size (opposite direction),
                // Both asteroids destroy each other, pop the top asteroid from the stack.
                if (!stack.empty() && stack.back() == -asteroid) {
                    stack.pop_back();
                } else if (stack.empty() || stack.back() < 0) {
                    // If the stack is empty or the top asteroid is moving to the left (negative),
                    // Add the current asteroid to the stack, since no collision will happen.
                    stack.push_back(asteroid);
                } // If none of the above cases, the current asteroid is destroyed
            }
        }
        return stack; // Return the state of the stack after all collisions
    }
};
```

Typescript Solution

```typescript
function asteroidCollision(asteroids: number[]): number[] {
    const stack: number[] = []; // Initialize an empty stack to store asteroids

    // Loop through each asteroid in the provided array
    for (const asteroid of asteroids) {
        // If the asteroid is moving to the right (positive), push it onto the stack
        if (asteroid > 0) {
            stack.push(asteroid);
        } else {
            // While there are still asteroids in the stack, and the top of the stack is moving right (positive)
            // and is smaller than the current asteroid (after negating its value),
            // pop the smaller asteroids from the stack as they will collide and explode
            while (stack.length && stack[stack.length - 1] > 0 && stack[stack.length - 1] < -asteroid) {
                stack.pop();
            }

            // If the top of the stack is an asteroid of the same size (after negating the current asteroid's value),
            // they will collide and explode, so pop the top of the stack
            if (stack.length && stack[stack.length - 1] === -asteroid) {
                stack.pop();
            } else if (stack.length === 0 || stack[stack.length - 1] < 0) {
                // If the stack is empty or the top of the stack is moving left (negative),
                // push the current asteroid onto the stack as there is no collision
                stack.push(asteroid);
            }
        }
    }

    // Return the stack which now contains the state of asteroids after all collisions
    return stack;
}
```

Time and Space Complexity

The time complexity of the provided code is `O(n)`, where `n` is the length of the array `asteroids`. This is due to the fact that the algorithm processes each element of the array exactly once in the worst-case scenario. The `while` loop inside the `for` loop does not increase the time complexity because it only processes elements that are potentially colliding, and each element can be pushed and popped from the stack at most once.

The space complexity of the code is also `O(n)`. In the worst-case scenario, this happens when all the asteroids are moving in the same direction and there are no collisions. As such, the `stk` can potentially grow to include all elements of the `asteroids` array if no asteroids ever collide.