838. Push Dominoes

Medium | Two Pointers | String | Dynamic Programming | Leetcode Link

Problem Description

In this problem, we're given a line of dominoes represented by a string of characters, with each character indicating the state of a domino. A domino can either be standing vertically (`.`), pushed to the left (`L`), or pushed to the right (`R`). The string represents the initial configuration of the dominoes at time zero.

When dominoes are pushed, they start falling and can cause adjacent dominoes to fall as well. If a domino is pushed to the left, it will make its left neighbor fall to the left in the next second, and similarly for the right. However, if dominoes are falling towards each other, they will make the domino in between them stand still (`.`) as the forces from both sides will cancel.

The objective is to determine the final state of all dominoes after all movements have stopped – which is when no further dominoes are being pushed over by their neighbors.

Intuition

The solution approach deals with the propagation of the falling action among dominoes. A key observation is to keep track of the time it takes for each domino to fall and the direction of the force applied to it. By initializing a queue to store the indices of dominoes that have been pushed, we can process each one by one.

We use Breadth-First Search (BFS) to simulate the domino effect. BFS is useful for problems where we need to propagate information (or in this case, force) level by level, from neighbor to neighbor. We start by adding indices with 'L' or 'R' to the queue and setting their time to 0 since these are the starting points of the falling process.

As we dequeue an index, we apply its force to its respective neighbor if the conditions allow - that is if the neighbor hasn't been pushed (`time[j]` is `-1`) or at the same time from the opposite side (`time[j] == t + 1`). If a domino receives force from both sides at the same time, it will remain standing – which is ensured by storing all forces in a `force` list and checking its length. When a domino gets only one direction of force, we proceed with the domino effect in that direction, queuing up the next domino in line.

In the end, we join and return the resulted string, which is the final state of the dominoes.

Solution Approach

The implementation uses a combination of Breadth-First Search (BFS), a queue, a `time` array, and a `force` dictionary – each element playing a specific role in simulating the domino effect.

Data Structures:

- **Queue (`q`)**: A double-ended queue `deque` is used for BFS to store the indices of the dominoes that are currently falling. It allows us to easily add new dominoes that start falling and process dominoes in the order they were pushed.
- **Time Array (`time`)**: An array that holds the time at which each domino falls. If a domino has not fallen, it is marked with `-1`. This helps in determining if a neighboring domino should be pushed (if it is still untouched) or to detect simultaneous pushes (a domino pushed at the same time from both sides).
- **Force Dictionary (`force`)**: A defaultdict with list values used to track all forces acting on a domino. If a domino is pushed at the same time from both sides, this can be detected when the length of the list for that index is greater than 1.

Algorithm:

1. Iterate through the `dominoes` string, and for each domino that is not upright (`'.'`), initialize the `time` at this index to `0` and add the forces acting on it to the `force` dictionary. Also, each such domino's index is added to the `q`.
2. Start processing the queue until it's empty:
   - Dequeue an index `i` and check the forces acting on it.
   - If there's only one force (`len(force[i]) == 1`):
     - Set the `ans` array at position `i` to the current force `f`.
     - Calculate the index `j` of the adjacent domino to push based on the current force's direction (`j = i - 1` if `f == 'L'` and `j = i + 1` if `f == 'R'`).
     - If `j` is within bounds:
       - If the neighbor has not been pushed yet (`time[j] == -1`), update its `time` to `t + 1`, add its force, and enqueue index `j`.
       - If the neighbor is being pushed at the same time (`time[j] == t + 1`), add the force to it.
3. Once the queue is empty, join the `ans` array into a string representing the final state of the dominoes and return it.

By following this approach, the algorithm simulates the force propagation, domino interaction, and stabilization to output the final arrangement of dominoes.

Example Walkthrough

Let's illustrate the solution approach with a small example, using the initial string of dominoes `..R...L..`. Here's how the algorithm processes this string:

1. Initialize data structures:
   - `dominoes`: `..R...L..`
   - `q`: []
   - `time`: `[-1, -1, 0, -1, -1, -1, 0, -1, -1]`
   - `force`: `{2: ['R'], 6: ['L']}`
   - `ans`: `['.', '.', '.', '.', '.', '.', '.', '.', '.']`
2. Fill the queue with indices of pushed dominoes and their respective times and forces:
   - The domino at index 2 is pushed to the right, so we add `2` to `q`.
   - The domino at index 6 is pushed to the left, so we add `6` to `q`.
3. Process the queue:
   - Dequeue index `2`, the force is `['R']`, so we push the domino at index `3`.
   - Dequeue index `6`, the force is `['L']`, so we push the domino at index `5`.
4. Continue the BFS:
   - At index `3`, since there was no push from the left, the domino falls to the right and we enqueue index `4` with a time of `1` (previous time + 1).
   - At index `5`, since there is no push from the right, the domino falls to the left and we enqueue index `4` with a time of `1`.
5. Resolve conflicts:
   - At index `4`, we have enqueued this index twice (once from the right and once from the left) both with time `1`. When dequeued, the `force` dictionary shows two forces acting on this domino, `['R', 'L']`, so it stays upright.
6. Output the final state:
   - Since there are no more dominoes to process, we concatenate our finalized `ans` array to get the final state of the dominoes: `..RR.L....`

In this way, using BFS and the associated data structures, we've efficiently simulated the entire domino chain-reaction and have arrived at the final arrangement without any iterative comparison.

Python Solution

```python
from collections import deque, defaultdict

class Solution:
    def pushDominoes(self, dominoes: str) -> str:
        # Length of the dominoes string
        num_dominoes = len(dominoes)

        # Queue to hold the indices of dominoes to process
        queue = deque()

        # Array to hold the time when each dominoe falls
        fall_time = [-1] * num_dominoes

        # Dictionary to hold the force(s) acting on each dominoe
        forces = defaultdict(list)

        # Initialize the queue, fall_time, and forces
        for index, force in enumerate(dominoes):
            if force != '.':
                queue.append(index)
                fall_time[index] = 0
                forces[index].append(force)

        # Resultant dominoes state
        result = ['.'] * num_dominoes

        # Process the queue
        while queue:
            current_idx = queue.popleft()
            if len(forces[current_idx]) == 1:
                resultant_force = forces[current_idx][0]
                result[current_idx] = resultant_force
                next_idx = current_idx - 1 if resultant_force == 'L' else current_idx + 1
                if 0 <= next_idx < num_dominoes:
                    current_time = fall_time[current_idx]
                    if fall_time[next_idx] == -1:
                        queue.append(next_idx)
                        fall_time[next_idx] = current_time + 1
                        forces[next_idx].append(resultant_force)
                    elif fall_time[next_idx] == current_time + 1:
                        forces[next_idx].append(resultant_force)

        # Return the final dominoes state as a string
        return ''.join(result)

# The code now uses more standard and descriptive variable names.
# Additionally, the comments provide a clear explanation of each part of the algorithm.

```

Java Solution

```java
public class Solution {
    public String pushDominoes(String dominoes) {
        int length = dominoes.length();
        // Queue to keep track of indices in the dominoes string that are affected by forces
        Deque<Integer> queue = new ArrayDeque<>();
        // Array to keep track of the time when each force reaches an index
        int[] times = new int[length];
        // Initialize all times to -1 (indicating no force has reached the index)
        Arrays.fill(times, -1);
        // List to store forces affecting each index (as multiple forces can affect the same index)
        List<Character>[] forces = new List[length];
        for (int i = 0; i < length; ++i) {
            forces[i] = new ArrayList<>();
        }

        // Initialize the forces and times with the initial state. Also add indices to queue to process
        for (int i = 0; i < length; ++i) {
            char force = dominoes.charAt(i);
            if (force != '.') {
                queue.offer(i);
                times[i] = 0;
                forces[i].add(force);
            }
        }

        // Array to store the final state of the dominoes
        char[] result = new char[length];
        Arrays.fill(result, '.');

        // Process the queue until it is empty
        while (!queue.isEmpty()) {
            int idx = queue.poll();
            // If only one force is affecting the index, update the result
            if (forces[idx].size() == 1) {
                char force = forces[idx].get(0);
                result[idx] = force;
                int nextIdx = force == 'L' ? idx - 1 : idx + 1;
                if (nextIdx >= 0 && nextIdx < length) {
                    int currentTime = times[idx];
                    if (times[nextIdx] == -1) {
                        queue.offer(nextIdx);
                        times[nextIdx] = currentTime + 1;
                        forces[nextIdx].add(force);
                    } else if (times[nextIdx] == currentTime + 1) {
                        forces[nextIdx].add(force);
                    }
                }
            }
        }

        // Convert char array to string and return
        return new String(result);
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    string pushDominoes(string dominoes) {
        int numDominoes = dominoes.size(); // Get the size of the string representing the dominoes.

        // Queue to keep track of indices of dominoes which have been tipped or will be tipped.
        queue<int> waitingQueue;

        // Vector to store the time at which each domino is tipped. Initialize with -1 (untipped).
        vector<int> tipTime(numDominoes, -1);

        // Vector of strings to store the forces acting on each domino.
        // Multiple forces can only act during the same time unit, hence using strings.
        vector<string> forces(numDominoes);

        for (int i = 0; i < numDominoes; i++) {
            if (dominoes[i] == '.') continue; // Skip if the domino has not been tipped.
            waitingQueue.emplace(i); // Add the index to the queue.
            tipTime[i] = 0; // Tipped at time 0 (initial state)
            forces[i].push_back(dominoes[i]); // Record the initial force ('L' or 'R')
        }

        // String to store the final state of the dominoes.
        string finalState(numDominoes, '.');

        // Process each domino in the queue.
        while (!waitingQueue.empty()) {
            int index = waitingQueue.front(); // Get the current index to be processed.
            waitingQueue.pop();

            // If only one force is acting on the domino, process it.
            if (forces[index].size() == 1) {
                char forceDirection = forces[index][0]; // Get the force direction 'L' or 'R'.
                finalState[index] = forceDirection; // Tip the domino in the final state.

                // Calculate the adjacent index. Left for 'L', Right for 'R'.
                int adjacentIndex = (forceDirection == 'L') ? (index - 1) : (index + 1);

                // Check if the adjacent index is within bounds.
                if (adjacentIndex >= 0 && adjacentIndex < numDominoes) {
                    int currentTime = tipTime[index]; // Get the current time.

                    // Tip the next domino if it hasn't been tipped yet.
                    if (tipTime[adjacentIndex] == -1) {
                        waitingQueue.emplace(adjacentIndex); // Add index to the queue.
                        tipTime[adjacentIndex] = currentTime + 1; // Update the time.
                        forces[adjacentIndex].push_back(forceDirection); // Apply the force.
                    // If the next domino is tipped at the same time, it means forces are colliding.
                    } else if (tipTime[adjacentIndex] == currentTime + 1)
                        forces[adjacentIndex].push_back(forceDirection); // Forces collide.
                }
            }
            // Collision case is implicitly handled by not tipping the domino in the final state.
        }

        return finalState; // Return the final state of the dominoes
    }
};

```

Typescript Solution

```typescript
function pushDominoes(dominoes: string): string {
    const length = dominoes.length;
    // Mapping the input characters to numerical values for easy processing
    const dominoMap = {
        'L': -1,
        'R': 1,
        '.': 0,
    };
    // Initialize the result array with numerical values
    let result = new Array(length).fill(0);
    // Track visited positions with their propagation depth
    let visited = new Array(length).fill(0);
    // Queue for BFS (Breadth-First Search) to track dominos to process
    let queue: number[] = [];
    // Start with depth 1 (first level of BFS)
    let currentDepth = 1;

    // Set up the queue with initial domino positions
    for (let index = 0; index < length; index++) {
        let dominoValue = dominoMap[dominoes.charAt(index)];
        if (dominoValue) {
            queue.push(index);
            visited[index] = currentDepth;
            result[index] = dominoValue;
        }
    }

    // Process the queue using BFS
    while (queue.length) {
        currentDepth++;
        let nextLevel: number[] = [];
        for (let position of queue) {
            const direction = result[position];
            let newPosition = position + direction;
            // If not out of bounds and not visited at current level or blocked
            if (newPosition >= 0 && newPosition < length && [0, currentDepth].includes(visited[newPosition])) {
                result[newPosition] += direction;
                visited[newPosition] = currentDepth;
                nextLevel.push(newPosition);
            }
        }
        queue = nextLevel;
    }

    // Convert the numerical result back into domino state characters
    return result
        .map(value => {
            if (value === 0) return '.';
            else if (value < 0) return 'L';
            else return 'R';
        })
        .join('');
}

```

Time and Space Complexity

The given code simulates the process of pushing dominoes. Let `n` be the length of the string `dominoes`. The time complexity and space complexity analysis are as follows:

Time Complexity:

1. The initialization of `time` and `force`, as well as the enumeration of `dominoes` to populate the queue `q`: O(n), since every domino is processed once.
2. The while loop processes each domino at most twice - once for each possible push direction ('L' or 'R'). The main processing within the while loop executes in constant time for each element, except for the deque operations.
3. The `popleft` operation from deque `q` is O(1) amortized per element.
4. Checking and updating the `force` at index `j` is also O(1) for each element since list append and length check are constant time operations.
5. Hence, every element can be inserted into the queue at most twice, once for being pushed to the left, and once for being pushed to the right.

The overall time complexity of the loop is O(n), because each element is dealt with in constant time and the queue ensures that each element is processed at most twice. Therefore, the total time complexity is `O(n) + O(n) = O(n)`.

Space Complexity:

1. The `time` list, `force` dictionary, and `ans` list each require O(n) space.
2. The `q` can have at most `n` elements in the worst case when all dominoes are getting pushed.
3. The `force` dictionary has lists, but each index `i` in `force` will have at most two forces (pushed from left and right) if dominoes at index `i` falls due to being pushed by both sides. So, space required by `force` values (lists) still remains within O(n) overall.

Thus, the space complexity is `O(n) + O(n) + O(n) = O(n)`.