1824. Minimum Sideway Jumps

Medium | Greedy | Array | Dynamic Programming

Problem Description

In this problem, we have a road that has exactly three lanes and extends from point 0 to point n. Picture the road as a grid where the horizontal axis represents the points 0 to n and the vertical axis represents the 3 lanes.

A frog starts on the second lane at point 0 and aims to get to any lane at point n. Along the way, the frog might encounter obstacles. The obstacles array contains `n+1` elements where each element indicates if there's an obstacle in a specific lane at a specific point. If `obstacles[i]` is 0, there are no obstacles at point `i`; otherwise, the value of `obstacles[i]` signifies which lane has an obstacle at point `i`.

The frog can only move forward from a point `i` to point `i+1` if there isn't any obstacle on the lane at point `i+1`. In addition, the frog can jump to another lane at the same point without moving forward, but only if there's no obstacle in the destination lane. Note that such sideways jumps are not free; our goal is to help the frog reach point `n` with the minimum number of these side jumps.

Here are the key points:

- The frog starts in the second lane at point 0.
- Obstacles are represented in an array, with `obstacles[i]` indicating an obstacle on lane `obstacles[i]` at point `i` (`obstacles[i]` ranges from 0 to 3, where 0 means no obstacle).
- The frog can move forward on the same lane or jump to a different lane at the same point (sideways).
- There are never any obstacles at the start or end points (0 and n).
- We need to calculate the minimum number of side jumps needed for the frog to reach any lane at point `n`.

Intuition

To solve this problem, we need to track the frog's potential positions and the number of jumps needed to reach those positions. We can do this by maintaining an array where each element corresponds to a lane and holds the count of the minimum number of side jumps needed to reach that lane at the current point.

Here's the approach step by step:

1. Initialize an array `f` with three elements, one for each lane, with the initial values set to represent the number of side jumps needed to start in the middle lane (which would be 0 for lane 2 and 1 for lanes 1 and 3 since the frog starts in lane 2 and would need one jump to reach the others).
2. Begin iterating through each point on the road, which are represented by the elements of the obstacles array excluding the first element (since there's no obstacle at point 0).
3. For each point, check if there's an obstacle in a lane. If there is, mark the corresponding element in `f` as infinity (`inf`) to represent that the lane is blocked and therefore cannot be the minimum.
4. Find the minimum value in `f` and increment it by 1. This accounts for the possibility of a side jump to avoid the current obstacle.
5. Update the values in `f` for lanes without obstacles at the current point. Their values are set to either their current value or the incremented minimum from the previous step, whichever is lower. This represents the minimum side jumps to reach that lane up to the current point, either by being there already or by making a jump.
6. After processing all points, the minimum value in the modified `f` array will represent the minimum number of side jumps to reach any lane at the destination point `n`.

Following this strategy will dynamically update the minimum jumps needed for each lane at each point, taking into consideration the possibility of avoiding obstacles by side jumping, leading us to the solution of the minimum side jumps throughout the entire road.

Solution Approach

The Python solution provided defines a class `Solution` with a method `minSideJumps`, which takes the `obstacles` array as an argument. This method implements dynamic programming to compute the minimum number of side jumps the frog needs to reach the end of the road.

1. **Dynamic Array Initialization**: A list `f` with three elements is initialized to `[1, 0, 1]`. These elements correspond to the number of side jumps needed to reach lanes 1, 2, and 3 respectively from the starting position (lane 2 at point 0). The frog is starting in the middle lane, so it requires no side jumps (`f[1] = 0`), but would require a side jump to get from the middle lane to either of the outer ones initially (`f[0]` and `f[2]` initialized to `1`).
2. **Iterating Over Obstacles**: The method then iterates through the `obstacles` array, starting at index 1 until the end (`obstacles[1:]`). This skips the starting point where we're guaranteed to have no obstacles.
3. **Updating Array per Obstacle**: For each point `i`, the list `f` is updated as follows:
   - If there's an obstacle in lane `j`, set `f[j]` to infinity (`inf`) to indicate that this lane is no longer an option without making an additional jump.
   - Calculate the minimum number of side jumps needed up to this point, which is `min(f) + 1`. The `+ 1` accounts for potentially having to jump away from an obstacle in the current position.
4. **Updating Array for Movement Options**: The list `f` is then updated again for each lane, ensuring that it represents the minimum number of side jumps to be at each lane at the current point:
   - If there's no obstacle in lane `j`, set `f[j]` to the minimum of its current value and the new calculated value `x`.
5. **Return Minimum Jumps to Reach End**: After all obstacles have been processed, the method returns the minimum value in `f`, which represents the minimum number of side jumps the frog needs to reach the end, across all three lanes.

This approach uses the *dynamic programming* pattern, where the `f` list serves as a memoization storage to keep track of the optimal number of side jumps up to each point. The solution leverages the concept of updating potential future states based on current state and the present obstacle configuration at each point.

The overall algorithm complexity is O(n), where n is the number of points (`n+1` points for a road of length `n`), because it processes each point exactly once, doing a constant amount of work for each.

Example Walkthrough

Let's consider a small example to illustrate the solution approach. Assume we have the road from point 0 to point n, with `n = 5`, and the obstacles are represented by the following array: `[0,1,0,2,3,0]`. This means:

- There are no obstacles at point 0 and point 5.
- At point 1, the obstacle is in lane 1.
- There's no obstacle at point 2.
- At point 3, the obstacle is in lane 2.
- At point 4, the obstacle is in lane 3.

The frog starts in lane 2 at point 0 and wants to end up at any lane at point 5 with the least side jumps possible.

1. **Dynamic Array Initialization**: We start with the list `f = [1, 0, 1]`, which is the number of jumps needed to get to lanes 1, 2, and 3 respectively from lane 2 at point 0.
2. **Iterating Over Obstacles**: We start iterating over the obstacles, skipping the first one because there's no obstacle at the starting point.
3. **First Obstacle**: We encounter the first obstacle in lane 1 at point 1. We set `f[0]` to infinity. Now `f = [inf, 0, 1]`.
4. **Minimum Jump Calculation**: Look for the minimum number of side jumps which is `min(inf, 0, 1)` and the result is 0. Add 1 for potential side jump so, `min_jump = 0 + 1`.
5. **Update For Movement Options**: Since lanes 2 and 3 are clear at point 1, set `f[1]` and `f[2]` to the minimum of their current value and `min_jump`, thus remaining as `f = [inf, 0, 1]`.
6. **No Obstacle at Point 2**: No updates since `obstacles[2]` is 0.
7. **Second Obstacle**: We encounter an obstacle in lane 2 at point 3. We set `f[1]` to infinity. Now `f = [inf, inf, 1]`.
8. **New Minimum Jump Calculation**: Only lane 3 is available with the current minimum of 1, so `min_jump = 1 + 1 = 2`.
9. **Update For Movement Options**: Only lane 3 is available; no change occurs as `f = [inf, inf, 1]`.
10. **Third Obstacle**: We encounter an obstacle in lane 3 at point 4. We set `f[2]` to infinity. Now `f = [inf, inf, inf]`.
11. **Final Minimum Jump Calculation**: This doesn't affect the minimum jump count since this happens at the last point, and lane 2 doesn't need additional jumps to reach the end.
12. **Return Minimum Jumps to Reach End**: The minimum number of jumps needed for the frog to finish is still 1, which is the minimum value in `f` for the lanes without infinity.

Upon reaching point 5, any lane will do, and the minimum number of side jumps the frog needed to make to get there in this example was 1. This helps us to visualize how the solution adapts to the dynamic array `f` as it processes the obstacles on the road.

Solution Implementation

Python

```python
class Solution:
    def minSideJumps(self, obstacles: List[int]) -> int:
        # Initialize the number of side jumps for each lane at the start.
        # Lane 2 has 0 jumps (since the frog starts here), others have 1 jump.
        jumps_count = [1, 0, 1]

        # Iterate through the obstacle positions, starting from the second position.
        for obstacle_position in obstacles[1:]:
            # Update the jump count to infinity for the lane with the obstacle.
            for lane in range(3):
                if obstacle_position == lane + 1:
                    jumps_count[lane] = float('inf')
                    break  # No need to check other lanes after finding the obstacle.

            # Find the minimum jumps needed plus one for the potential jump.
            min_jumps_plus_one = min(jumps_count) + 1

            # Update the jump counts for lanes that do not have the current obstacle.
            for lane in range(3):
                if obstacle_position != lane + 1:
                    jumps_count[lane] = min(jumps_count[lane], min_jumps_plus_one)

        # Return the minimum jumps count from the updated jump counts.
        return min(jumps_count)
```

Java

```java
class Solution {
    // Method to find the minimum number of side jumps to reach the end of the array.
    public int minSideJumps(int[] obstacles) {
        final int INF = Integer.MAX_VALUE; // Use a large number to represent an 'infinite' number of jumps.
        int[] jumps = {1, 0, 1}; // Array to store the minimum jumps needed to reach each lane at the current point.

        // Iterate over the obstacle array, skipping the first element as the starting point is fixed.
        for (int position = 1; position < obstacles.length; ++position) {
            // Update the number of jumps to INF if there is an obstacle in the lane.
            for (int lane = 0; lane < 3; ++lane) {
                // If there is an obstacle in the current lane, set jumps for that lane to infinity.
                if (obstacles[position] == lane + 1) {
                    jumps[lane] = INF;
                }
            }

            // Find the minimum jumps required to reach any lane without an obstacle placed on position.
            int minJumpsUntilNow = Math.min(jumps[0], Math.min(jumps[1], jumps[2])) + 1;

            // Update the number of jumps for lanes with no obstacles at the current position.
            for (int lane = 0; lane < 3; ++lane) {
                // If there is no obstacle in the lane, it could be beneficial to jump to this lane.
                if (obstacles[position] != lane + 1) {
                    jumps[lane] = Math.min(jumps[lane], minJumpsUntilNow);
                }
            }
        }

        // Return the minimum of the jumps needed to be in any of the three lanes at the last position.
        return Math.min(jumps[0], Math.min(jumps[1], jumps[2]));
    }
}
```

C++

```cpp
#include <vector>
#include <algorithm> // for std::min

class Solution {
public:
    int minSideJumps(vector<int>& obstacles) {
        const int INF = 1 << 30; // Use a large number to represent infinity
        int lanes[3] = {1, 0, 1}; // Initialize the side jumps required for each lane

        // Loop through each position on the race track
        for (int position = 1; position < obstacles.size(); ++position) {
            // Update lane status based on obstacles
            for (int lane = 0; lane < 3; ++lane) {
                // If there's an obstacle in the lane, set jump count to infinity
                if (obstacles[position] == lane + 1) {
                    lanes[lane] = INF;
                    break;
                }
            }
            // Find the minimum number of side jumps required to reach this position
            int minJumps = std::min({lanes[0], lanes[1], lanes[2]}) + 1;

            // Update the jumps needed for now obstacle-free lanes
            for (int lane = 0; lane < 3; ++lane) {
                // If there's no obstacle in the lane, take the min of the current
                // and the just computed min jumps needed to reach this position
                if (obstacles[position] != lane + 1) {
                    lanes[lane] = std::min(lanes[lane], minJumps);
                }
            }
        }
        // Return the minimum number of jumps among the three lanes
        return std::min({lanes[0], lanes[1], lanes[2]});
    }
};
```

TypeScript

```typescript
function minSideJumps(obstacles: number[]): number {
    // Initialize a very large number to represent an infinite cost, used for unreachable paths
    const infinityCost = 1 << 30;
    // f represents the minimum jumps needed to reach each lane (starting from lane 2)
    let jumps = [1, 0, 1];

    // Iterate over the obstacle positions
    for (let position = 1; position < obstacles.length; ++position) {
        // Set unreachable lanes to infinity cost due to an obstacle
        for (let lane = 0; lane < 3; ++lane) {
            if (obstacles[position] == lane + 1) {
                jumps[lane] = infinityCost;
                break;
            }
        }

        // Calculate the additional cost to switch to a different lane if necessary
        const minJumpsPlusOne = Math.min(...jumps) + 1;

        for (let lane = 0; lane < 3; ++lane) {
            // Update the minimum jumps for reachable lanes to the smallest value
            // of staying or switching lanes
            if (obstacles[position] != lane + 1) {
                jumps[lane] = Math.min(jumps[lane], minJumpsPlusOne);
            }
        }
    }

    // Return the minimum jumps needed to reach the end for all lanes
    return Math.min(...jumps);
}
```

```python
class Solution:
    def minSideJumps(self, obstacles: List[int]) -> int:
        # Initialize the number of side jumps for each lane at the start.
        # Lane 2 has 0 jumps (since the frog starts here), others have 1 jump.
        jumps_count = [1, 0, 1]

        # Iterate through the obstacle positions, starting from the second position.
        for obstacle_position in obstacles[1:]:
            # Update the jump count to infinity for the lane with the obstacle.
            for lane in range(3):
                if obstacle_position == lane + 1:
                    jumps_count[lane] = float('inf')
                    break  # No need to check other lanes after finding the obstacle.

            # Find the minimum jumps needed plus one for the potential jump.
            min_jumps_plus_one = min(jumps_count) + 1

            # Update the jump counts for lanes that do not have the current obstacle.
            for lane in range(3):
                if obstacle_position != lane + 1:
                    jumps_count[lane] = min(jumps_count[lane], min_jumps_plus_one)

        # Return the minimum jumps count from the updated jump counts.
        return min(jumps_count)
```

Time and Space Complexity

Time Complexity:

The time complexity of the provided code is `O(n)` where `n` is the number of elements in the `obstacles` list. The algorithm iterates through each of the `n` obstacles once. Within this loop, it performs constant-time operations - updating the `f` array and checking conditions. It does not use nesting of loops; hence other than the linear pass through the `obstacles`, there aren't additional multiplicative factors contributing to the time complexity.

Space Complexity:

The space complexity of the provided code is `O(1)`. This is because the space used by the algorithm does not scale with the input size `n`. The `f` array is of a fixed size of 3, representing the number of lanes. There are no other data structures used that grow with the input size. Thus, regardless of the size of the `obstacles` list, the amount of space used by the algorithm remains constant.