1345. Jump Game IV

Hard · Breadth-First Search · Array · Hash Table · Leetcode Link

Problem Description

The problem involves an array of integers `arr` and a character that starts at the first index of the array. This character can jump to a new index in one of three ways. They can jump to the next index (`i + 1`), jump back to the previous index (`i - 1`), or jump to any index `j` that contains the same value as the current index (`arr[i] == arr[j]` and `i != j`). The goal is to determine the minimum number of steps required to reach the last index of the array. It is important to note that jumps can only occur within the bounds of the array, meaning that the character cannot jump to an index outside of the array.

Intuition

To arrive at the solution, we use breadth-first search (BFS) as the main strategy since the goal is to find the minimum number of steps. BFS is suitable for this kind of problem because it explores the nearest neighbors first and only goes deeper once there are no more neighbors to explore. Given this approach, we start from the first index and explore all possible jumps. To avoid revisiting indices and getting stuck in loops, we use a visited set (`vis`) to keep track of which indices have already been explored. Also, to avoid repeatedly performing the same jumps, we use a dictionary (`idx`) that maps each value to its indices in the array.

As we progress, we store in a queue (`q`) each index we can jump to, along with the number of steps taken to reach there. Once the jump to the last index is made, we return the number of steps that led to it as the result. To make the BFS efficient, after checking all indices that have the same value as the current index, we delete the entry from the `idx` dictionary to prevent redundant jumps in the future. This speeds up the BFS process significantly, as it reduces the number of potential jumps that are not helpful in reaching the end of the array quickly.

Solution Approach

To solve this problem, the code uses the Breadth-First Search (BFS) algorithm, which is a standard approach for finding the shortest path in an unweighted graph or the fewest number of steps to reach a certain point.

Here's a walkthrough of the implementation:

1. **Index Dictionary (`idx`)**: A `defaultdict` from Python's collections module is used to create a dictionary where each value in the array is a key, and the corresponding value is a list of all indices in the array with that value.

   ```python
   idx = defaultdict(list)
   for i, v in enumerate(arr):
       idx[v].append(i)
   ```

2. **Queue (`q`)**: A queue is implemented with `deque` from the collections module, and it's initialized with a tuple containing the starting index (`0`) and the initial step count (`0`).

   ```python
   q = deque([(0, 0)])
   ```

3. **Visited Set (`vis`)**: A set is used to keep track of the indices that have been visited. This prevents the algorithm from processing an index more than once and ensures not to get caught in an infinite loop.

   ```python
   vis = {0}
   ```

4. **BFS Loop**: The algorithm processes the queue `q` until it's empty, doing the following:

   - It dequeues an element, which is a tuple of the current index (`i`) and the number of steps taken to reach this index (`step`).
   - If the dequeued index is the last index of the array, the number of steps is immediately returned.
   - All indices `j` in the array where `arr[i] == arr[j]` are added to the queue, with the number of steps incremented by `1`, and marked as visited.
   - After processing all jumps to the same value, that value is deleted from `idx` to prevent future unnecessary jumps to those indices.
   - The algorithm then looks to add the neighbor indices `i + 1` and `i - 1` to the queue (if they haven't been visited yet).

   ```python
   while q:
       i, step = q.popleft()
       if i == len(arr) - 1:
           return step
       v = arr[i]
       step += 1
       for j in idx[v]:
           if j not in vis:
               vis.add(j)
               q.append((j, step))
       del idx[v]
       if i + 1 < len(arr) and (i + 1) not in vis:
           vis.add(i + 1)
           q.append((i + 1, step))
       if i - 1 >= 0 and (i - 1) not in vis:
           vis.add(i - 1)
           q.append((i - 1, step))
   ```

By employing BFS, which explores all possible paths breadthwise and visits each node exactly once, the solution finds the minimum number of steps needed to reach the last index of the array.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach for the array `arr = [100, 23, 23, 100, 23, 100]`.

1. **Initialize the Index Dictionary (`idx`)**: We create a dictionary mapping each value in the array to a list of all indices in the array with that value.
   - `idx[100] = [0, 3, 5]`
   - `idx[23] = [1, 2, 4]`
2. **Initialize the Queue (`q`)**: We start with the first index and `0` steps as we haven't yet taken any steps.
   - `q = deque([(0, 0)])`
3. **Initialize the Visited Set (`vis`)**: To begin, we've only visited the start index `0`.
   - `vis = {0}`
4. **Begin the BFS Loop**:
   - Dequeue `(0, 0)`. Current index `i = 0`, steps taken `step = 0`.
     - We found the value `100` at index `0`. Indices with value `100` are `[0, 3, 5]`.
     - Queue up index `3` and `5` with `step+1` since `0` is already visited.
     - Update queue: `q = deque([(3, 1), (5, 1)])` and `vis = {0, 3, 5}`.
     - Remove `100` from the `idx` dictionary to prevent future jumps to these indices: `del idx[100]`.
   - Dequeue `(3, 1)`. Current index `i = 3`, steps taken `step = 1`.
     - Value at index `3` is `100`, but we've already deleted it from `idx`.
     - We check the neighbors, and `4` has not been visited, so we queue it up.
     - Update queue: `q = deque([(5, 1), (4, 2)])` and `vis = {0, 3, 4, 5}`.
   - Dequeue `(5, 1)`. Current index `i = 5`, steps taken `step = 1`.
     - We've reached the last index of the array, so we return the number of steps: `1`.

In this example, by using BFS and our optimizations, we've found that we can reach the end of the array `arr` in just `1` step through the value jumps. We can jump directly from the first index to the last index since they share the same value of `100`.

Python Solution

```python
from collections import defaultdict, deque
from typing import List

class Solution:
    def min_jumps(self, arr: List[int]) -> int:
        # Initialize a dictionary to keep track of indices for each value in the array
        index_map = defaultdict(list)
        for index, value in enumerate(arr):
            index_map[value].append(index)

        # Initialize a queue with a tuple containing the start index and initial step count
        queue = deque([(0, 0)])

        # Set to keep track of visited nodes to avoid cycles
        visited = set([0])

        # Process the queue until empty
        while queue:
            position, step_count = queue.popleft()

            # If the end of the array is reached, return the count of steps
            if position == len(arr) - 1:
                return step_count

            # Increment the count of steps for the next jump
            step_count += 1

            # Jump to all indexes with the same value as the current position
            value = arr[position]
            for next_position in index_map[value]:
                if next_position not in visited:
                    visited.add(next_position)
                    queue.append((next_position, step_count))

            # Since all jumps for this value are done, clear the index list to prevent future jumps to the same value
            del index_map[value]

            # Add the next consecutive index to the queue if it hasn't been visited
            if position + 1 < len(arr) and (position + 1) not in visited:
                visited.add(position + 1)
                queue.append((position + 1, step_count))

            # Add the previous index to the queue if it hasn't been visited and it's within bounds
            if position - 1 >= 0 and (position - 1) not in visited:
                visited.add(position - 1)
                queue.append((position - 1, step_count))

        return -1  # If end is not reached, return -1 (this part of code should not be reached).

# Example usage:
# sol = Solution()
# print(sol.min_jumps([100, -23, -23, 404, 100, 23, 23, 23, 3, 404]))

```

Java Solution

```java
class Solution {

    // Finds minimum number of jumps to reach the end of the array.
    public int minJumps(int[] arr) {
        // Hash map to store indices of values in the array.
        Map<Integer, List<Integer>> indexMap = new HashMap<>();
        int n = arr.length;

        // Populate the hash map with indices for each value.
        for (int i = 0; i < n; ++i) {
            indexMap.computeIfAbsent(arr[i], k -> new ArrayList<>()).add(i);
        }

        // Queue for BFS, each element is a pair: [current index, current step count].
        Deque<int[]> queue = new LinkedList<>();
        // Set to keep track of visited indices.
        Set<Integer> visited = new HashSet<>();

        // Start BFS with the first index.
        visited.add(0);
        queue.offer(new int[] {0, 0});

        // BFS to find minimum steps.
        while (!queue.isEmpty()) {
            int[] element = queue.pollFirst();
            int currentIndex = element[0], stepCount = element[1];

            // If we've reached the end of the array, return the step count.
            if (currentIndex == n - 1) {
                return stepCount;
            }

            // Increment step count for next potential moves.
            stepCount++;

            // Get all indices with the same value and add unseen ones to the queue.
            for (int index : indexMap.getOrDefault(arr[currentIndex], new ArrayList<>())) {
                if (!visited.contains(index)) {
                    visited.add(index);
                    queue.offer(new int[] {index, stepCount});
                }
            }

            // We remove this value from the map to prevent revisiting.
            indexMap.remove(arr[currentIndex]);

            // Check and add unseen next and previous indices to the queue.
            if (currentIndex + 1 < n && !visited.contains(currentIndex + 1)) {
                visited.add(currentIndex + 1);
                queue.offer(new int[] {currentIndex + 1, stepCount});
            }
            if (currentIndex - 1 >= 0 && !visited.contains(currentIndex - 1)) {
                visited.add(currentIndex - 1);
                queue.offer(new int[] {currentIndex - 1, stepCount});
            }
        }

        // If we've exhausted all options and haven't reached the end, return -1.
        return -1;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_map>
#include <unordered_set>
#include <queue>

class Solution {
public:
    // Function to find the minimum number of jumps required to reach
    // the last index of the array where each element indicates the maximum
    // length of jump we can make from that position.
    int minJumps(std::vector<int>& arr) {
        // Map to hold the indices for each value in arr.
        std::unordered_map<int, std::vector<int>> indexMap;
        int n = arr.size();

        // Populate the index map with the indices for each value.
        for (int i = 0; i < n; ++i) {
            indexMap[arr[i]].push_back(i);
        }

        // Queue to perform BFS, holding pairs of the index in arr and the number of steps taken.
        std::queue<std::pair<int, int>> bfsQueue;
        bfsQueue.emplace(0, 0); // Start from the first element.

        // Set to keep track of visited indices to avoid revisits.
        std::unordered_set<int> visited;
        visited.insert(0);

        // Perform BFS.
        while (!bfsQueue.empty()) {
            auto current = bfsQueue.front();
            bfsQueue.pop();
            int currentIndex = current.first, steps = current.second;

            // If we have reached the last index, return the number of steps.
            if (currentIndex == n - 1) return steps;

            // Number of steps to reach the next index.
            ++steps;
            int currentValue = arr[currentIndex];

            // If we can jump to any index with the same value as currentValue.
            if (indexMap.count(currentValue)) {
                for (int nextIndex : indexMap[currentValue]) {
                    if (!visited.count(nextIndex)) {
                        visited.insert(nextIndex);
                        bfsQueue.emplace(nextIndex, steps);
                    }
                }
                // To avoid unnecessary iterations in the future, erase the value from the map.
                indexMap.erase(currentValue);
            }

            // Check the possibilities of jumping to adjacent indices.
            if (currentIndex + 1 < n && !visited.count(currentIndex + 1)) {
                visited.insert(currentIndex + 1);
                bfsQueue.emplace(currentIndex + 1, steps);
            }
            if (currentIndex - 1 >= 0 && !visited.count(currentIndex - 1)) {
                visited.insert(currentIndex - 1);
                bfsQueue.emplace(currentIndex - 1, steps);
            }
        }

        return -1; // If it's not possible to reach the end, return -1.
    }
};

```

Typescript Solution

```typescript
type IndexStepsPair = { index: number; steps: number };

function minJumps(arr: number[]): number {
  // Map to hold the indices for each value in arr.
  const indexMap: Record<number, number[]> = {};
  const n: number = arr.length;

  // Populate the index map with the indices for each value.
  for (let i = 0; i < n; i++) {
    if (!(arr[i] in indexMap)) {
      indexMap[arr[i]] = [];
    }
    indexMap[arr[i]].push(i);
  }

  // Queue to perform BFS, holding pairs of the index in arr and the number of steps taken.
  const bfsQueue: IndexStepsPair[] = [{ index: 0, steps: 0 }];

  // Set to keep track of visited indices to avoid revisits.
  const visited: Set<number> = new Set();
  visited.add(0);

  // Perform BFS.
  while (bfsQueue.length > 0) {
    const current = bfsQueue.shift()!; // Assumed to not be undefined
    const currentIndex = current.index;
    let steps = current.steps;

    // If we have reached the last index, return the number of steps.
    if (currentIndex === n - 1) {
      return steps;
    }

    // Number of steps to reach the next index.
    steps++;

    if (indexMap[arr[currentIndex]]) {
      for (const nextIndex of indexMap[arr[currentIndex]]) {
        if (!visited.has(nextIndex)) {
          visited.add(nextIndex);
          bfsQueue.push({ index: nextIndex, steps: steps });
        }
      }
      // To avoid unnecessary iterations in the future, delete the value from the map.
      delete indexMap[arr[currentIndex]];
    }

    // Check the possibilities of jumping to adjacent indices.
    if (currentIndex + 1 < n && !visited.has(currentIndex + 1)) {
      visited.add(currentIndex + 1);
      bfsQueue.push({ index: currentIndex + 1, steps: steps });
    }
    if (currentIndex - 1 >= 0 && !visited.has(currentIndex - 1)) {
      visited.add(currentIndex - 1);
      bfsQueue.push({ index: currentIndex - 1, steps: steps });
    }
  }

  // If it's not possible to reach the end, return -1.
  return -1;
}

```

Time and Space Complexity

The time complexity of the provided code is `O(N + E)` where `N` is the number of elements in `arr` and `E` is the total number of edges in the graph constructed where each value in `arr` has edges to indices with the same value. The while loop runs for each index in `arr` once, and each index is added to the queue at most once. The inner loops contribute to the edges. Deletion of keys in the `idx` dictionary ensures that each value's list is iterated over at most once, hence, contributing to `E`.

The space complexity is `O(N)` as we need to store the indices in the `idx` dictionary, which in the worst case will store all indices, and the `vis` set, which also could potentially store all indices if they are visited. The queue could also hold up to `N-1` elements in the worst case scenario.