565. Array Nesting

Medium · Depth-First Search · Array

Problem Description

In this problem, you are given an integer array `nums` with a length of `n`, where `nums` is a permutation of the numbers in the range `[0, n - 1]`. The goal is to create a unique set `s[k]` for each index `k` in the array. The set `s[k]` is formed by repeatedly applying the transformation `nums[i] -> nums[nums[i]]` starting with `nums[k]`. The transformation stops when it would add a duplicate element to `s[k]`.

In other words:

1. Begin with an index `k` and the corresponding element `nums[k]`, and add it to the set `s[k]`.
2. Find the next element by using the current element as the index to look up the next value in the array (`nums[nums[k]]`), and add that to the set.
3. Continue the process, each time using the latest element added to the set as the new index to look up.
4. Stop when you are about to add an element that is already in the set, because that would create a duplicate.

The task is to determine the size of the largest set `s[k]` that can be created by following the above rules.

Intuition

To tackle the problem, note that since `nums` is a permutation of the numbers in `[0, n - 1]`, every element in `nums` is part of exactly one cycle. Once you visit an element in `nums`, all subsequent elements that would be visited by following the transformation rule are part of the same cycle, and re-visiting them would just repeat the cycle without increasing the size of any set `s[k]`.

The strategy, therefore, is to iterate through each element of `nums` and trace out the cycle, marking elements as visited by setting them to a value outside the range `[0, n - 1]` (for instance, `n`). Count the number of steps it takes to complete the cycle. This count represents the size of the set `s[k]` formed by starting at the index `k`.

By doing this for each index `k` and keeping track of the maximum count found, we can identify the longest length of a set `s[k]`. The elements are marked as visited to avoid redundant computations and to ensure that once an element is a part of a set, it is not counted again for subsequent indices.

Solution Approach

The implemented solution follows a simple yet effective approach. It traverses through each element of the array `nums`, and for each element, it tracks the formation of the set `s[k]` by following the chain of indices as described by the transformation rule `nums[i] -> nums[nums[i]]`.

Here is the step-by-step explanation of the process using the given code:

1. Initialize a variable `ans` to store the length of the largest set `s[k]` and set its initial value to zero (`0`). Initialize `n` to the length of `nums`.
2. Loop through each index `i` from `0` to `n-1`. The variable `cnt` is used to keep track of the length of the set `s[k]` starting at index `i`.
3. Inside the loop, follow the cycle that starts at `nums[i]`. The while loop continues as long as the current element `nums[i]` has not been visited (a visited element is marked by setting it to `n`).
4. Retrieve the number at the current index `i` (stored in `j`), then mark the element at `i` as visited by setting `nums[i]` to `n`.
5. Set the current index `i` to `j` to move to the next number in the set.
6. Increment the counter `cnt` which reflects the size of the current set `s[k]`.
7. Once a cycle is traced (and the while loop exits), compare the size of this set (`cnt`) with the previous maximum (`ans`) and update `ans` if `cnt` is larger.
8. Continue with the next index, marking elements and counting the size of each set until all indices are processed.
9. Return `ans`, which is the length of the longest set `s[k]`.

Through this algorithm, all sets are explored only once, since any index that has been visited and marked will not be considered again. The code effectively avoids duplicating work by marking the visited elements. This mechanism is key to ensuring the algorithm runs efficiently and the solution's time complexity is kept at O(n), where `n` is the size of the input array `nums`.

Mathematically, you could view this as tracing cycles in a graph where nodes represent elements in the array and directed edges connect node `i` to node `nums[i]`. The objective is to find the length of the longest cycle in this graph without visiting any node more than once.

Example Walkthrough

Let's take an example array `nums` with the permutation of numbers `[1, 2, 0, 4, 5, 3]`. Here's the step-by-step walkthrough to find the size of the largest set `s[k]`.

1. Initialize the answer `ans` to `0`. Since the length of `nums` is `6`, initialize `n` to `6`.
2. Start the loop with index `i = 0`. Here, `nums[0] = 1`. Initialize `cnt = 0` which will count the size of set `s[0]`.
3. Begin the transformation for set `s[0]`:
   - `nums[0] = 1`, so the next index to visit is `1` and `cnt` becomes `1`. We mark `nums[0]` as visited by setting it to `n`.
   - Now, `i = 1` and `nums[1] = 2`, so the next index to visit is `2` and `cnt` becomes `2`. We mark `nums[1]` as visited.
   - Next, `i = 2` and `nums[2] = 0`, but `nums[0]` has already been visited, so the cycle is complete, and we stop here.
4. The size of set `s[0]` is `2`. We compare it with `ans` and since `2 > 0`, we update `ans` to `2`.
5. Move to the next index `i = 1`, but we have already visited `nums[1]`, so we skip to the next one, `i = 2`. This index is also visited, so we move on to `i = 3`.
6. For `i = 3`, the process is:
   - `nums[3] = 4` and `cnt` is reset to `1`. We mark `nums[3]` as visited.
   - Next, `i = 4` and `nums[4] = 5`. Increment `cnt` to `2` and mark `nums[4]` as visited.
   - Then, `i = 5` and `nums[5] = 3`. `cnt` is incremented to `3` and mark `nums[5]` as visited.
   - Finally, `nums[3]` is already marked as visited, so we end the cycle.
7. The size of the set formed starting at index `3` is `3`. We compare it with `ans`, and since `3 > 2`, we update `ans` to `3`.
8. Continuing this process for the remaining unvisited indices (which are none in this case), we find that no other set has a size larger than `3`.
9. Having completed the cycle for each index, the largest set `s[k]` has size `3`, so we return `ans = 3`.

Through the above example, we were able to find the longest cycle in the permutation graph, which represents the size of the largest set `s[k]`. The algorithm efficiently marks visited nodes to avoid redundant calculations, leading to an optimized solution with a linear time complexity relative to the size of the array `nums`.

Solution Implementation

Python

```python
from typing import List  # We need to import List from typing to use it as a type hint.

class Solution:
    def arrayNesting(self, nums: List[int]) -> int:
        # Initialize the max size of the nesting to 0.
        max_nesting_size = 0
        # Get the length of the input list.
        num_elements = len(nums)

        # Iterate through each element in the list.
        for i in range(num_elements):
            # Initialize count for the current set.
            current_set_size = 0
            # Continue traversing the set until we find an element that is marked as visited.
            while nums[i] != num_elements:
                # Fetch the next index from the current element.
                next_index = nums[i]
                # Mark the current element as visited by setting it to num_elements.
                nums[i] = num_elements
                # Move to the next index.
                i = next_index
                # Increment the size count for the current set.
                current_set_size += 1
            # Update the maximum nesting size if the current set size is larger.
            max_nesting_size = max(max_nesting_size, current_set_size)

        # Return the maximum size of nesting found.
        return max_nesting_size
```

Java

```java
class Solution {
    public int arrayNesting(int[] nums) {
        int maxNestSize = 0; // Variable to keep the size of the largest nest
        int arrayLength = nums.length; // Get the length of the array

        for (int start = 0; start < arrayLength; ++start) { // Loop through each element in the array
            int size = 0; // Initialize size for the current nest
            int currentIndex = start; // Starting index for the current nest

            // Iterate through the nest starting at currentIndex until a visited element is found
            while (nums[currentIndex] < arrayLength) { // Visited elements are marked with value equal or greater than arrayLengt
                int nextIndex = nums[currentIndex]; // Get the next index in the nest
                nums[currentIndex] = arrayLength; // Mark the current index as visited
                currentIndex = nextIndex; // Move to the next index
                ++size; // Increase the size of the current nest
            }

            maxNestSize = Math.max(maxNestSize, size); // Update the size of the largest nest found so far
        }

        return maxNestSize; // Return the size of the largest nest
    }
}
```

C++

```cpp
class Solution {
public:
    int arrayNesting(vector<int>& nums) {
        int maxNestSize = 0;                   // This will hold the maximum size of the nest
        int numElements = nums.size();         // Get the number of elements in the array

        // Iterate through each element in the array
        for (int i = 0; i < numElements; ++i) {
            int nestSize = 0;                  // Initialize the size for the current nest
            int currentIndex = i;              // Start nesting from the current index

            // Loop to find the nest size starting from the current index
            while (nums[currentIndex] < numElements) {
                int tempIndex = nums[currentIndex]; // Store next index from the current element's value
                nums[currentIndex] = numElements;   // Mark the current element as visited
                currentIndex = tempIndex;           // Move to the next index in the nest
                ++nestSize;                         // Increment the nest size
            }

            // Update maxNestSize if the current nest is bigger
            maxNestSize = max(maxNestSize, nestSize);
        }

        return maxNestSize;                   // Return the largest nest size found
    }
};
```

TypeScript

```typescript
function arrayNesting(nums: number[]): number {
    let maxNestSize: number = 0;  // This will hold the maximum size of the nest
    let numElements: number = nums.length;  // Get the number of elements in the array

    // Iterate through each element in the array
    for (let i = 0; i < numElements; ++i) {
        let nestSize: number = 0;  // Initialize the size for the current nest
        let currentIndex: number = i;  // Start nesting from the current index

        // Loop to find the nest size starting from the current index
        while (nums[currentIndex] < numElements) {
            let tempIndex: number = nums[currentIndex]; // Store the next index from the current element's value
            nums[currentIndex] = numElements;  // Mark the current element as visited
            currentIndex = tempIndex;  // Move to the next index in the nest
            nestSize++;  // Increment the nest size
        }

        // Update maxNestSize if the current nest is larger
        maxNestSize = Math.max(maxNestSize, nestSize);
    }

    return maxNestSize;  // Return the largest nest size found
}
```


```python
from typing import List  # We need to import List from typing to use it as a type hint.

class Solution:
    def arrayNesting(self, nums: List[int]) -> int:
        # Initialize the max size of the nesting to 0.
        max_nesting_size = 0
        # Get the length of the input list.
        num_elements = len(nums)

        # Iterate through each element in the list.
        for i in range(num_elements):
            # Initialize count for the current set.
            current_set_size = 0
            # Continue traversing the set until we find an element that is marked as visited.
            while nums[i] != num_elements:
                # Fetch the next index from the current element.
                next_index = nums[i]
                # Mark the current element as visited by setting it to num_elements.
                nums[i] = num_elements
                # Move to the next index.
                i = next_index
                # Increment the size count for the current set.
                current_set_size += 1
            # Update the maximum nesting size if the current set size is larger.
            max_nesting_size = max(max_nesting_size, current_set_size)

        # Return the maximum size of nesting found.
        return max_nesting_size
```


Time and Space Complexity

Time Complexity

The time complexity of the code is `O(n)`. This is because each element is visited only once. The `while` loop marks visited elements by setting their value to `n`, ensuring that each element can become the start of a set at most once. Since the input array has `n` elements, and we are doing constant time operations per element, traversing and marking all elements results in a linear time complexity relative to the size of the input array.

Space Complexity

The space complexity of the code is `O(1)` (ignoring the input size). This is due to the fact that no additional space proportional to the input size is used. The variables `ans`, `n`, `cnt`, `i`, and `j` only use a constant amount of extra space.