2459. Sort Array by Moving Items to Empty Space

Hard Greedy Array Sorting

Leetcode Link

Problem Description

In this problem, we are given an integer array `nums` of size `n` which contains each number from `0` to `n - 1` inclusive. Each number from `1` to `n - 1` represents a distinct item, while `0` represents an empty space. The objective is to determine the minimum number of moves needed to sort the array. Sorting the array means that the items must be in ascending order with the empty space (`0`) being at either the beginning or the end of the array. We can move any item into the empty space in one operation.

For example, with `n = 4`, the sorted arrays could be either `[0,1,2,3]` or `[1,2,3,0]`. Any other arrangement of these numbers is considered unsorted. The challenge is to find the minimum number of operations to reach a sorted state from the given unsorted state.

Intuition

To solve this problem, the solution harnesses the concept of the minimum number of swaps needed to sort a permutation when a single empty space can be used to facilitate the swaps. This scenario is similar to the well-known cycle-sort algorithm, with the unique twist of an empty space represented by `0`.

The intuition is that the array can be viewed as a graph where each item `i` points to the position `nums[i]`. There will be one or more cycles in this graph. To sort the array, each cycle of length `l` requires exactly `l - 1` swaps if no empty space is involved. The empty space enables us to save one swap per cycle if it is placed at an optimal location.

In the provided solution approach, there are two separate calls to the function `ff()` with the difference being where the empty space (`0`) is factored into the cycles: either at the beginning or at the end. This mirrors the two valid sorted configurations. By calculating cycles with empty space at the start or at the end, we find two different counts for the minimum operations needed and then we choose the smaller one.

The function `ff()` goes through each item in the array and counts the number of moves needed to place all items it reaches into their correct positions, looping through the cycles while marking visited items with a `vis` array to avoid recalculation. The number of moves needed for each cycle is decremented by 2 if the empty space is part of that cycle (`nums[k] != k`). Finally, the answer is the minimum number of moves required when considering the empty space at either the start (`k = 0`) or the end (`k = n - 1`).

It is important to note that the transformation `((v - 1 + n) % n for v in nums)` is done to simulate the empty space as if it were at the end of the array rather than at the beginning, since we cannot directly change the physical position of elements in the input array `nums`.

Solution Approach

The solution approach follows these key steps:

1. Define a helper function `f(nums, k)` that will be used to calculate the minimum number of operations needed to place all items in correct order with the empty space (`0`) located either at the beginning (`k = 0`) or at the end (`k = n - 1`) of the array.
2. Use a `vis` (visited) array to keep track of the elements that have already been moved into their correct positions. This prevents double counting of operations for items that are part of the same cycle.
3. Iterate through the array (using the variable `i`), identify cycles, and count the number of operations needed. This is done by:
   - Skip if `i == v` (the item is already at the correct position) or if `vis[i]` is `True` (item has already been visited).
   - Increment the `cnt` (number of operations needed) each time we visit a new item to be placed in the correct position.
   - Follow the cycle starting from the element at index `i` and mark elements as visited `vis[j] = True`. Continue the cycle until it closes back on itself.
4. Correct the operations count by subtracting `2 * (nums[k] != k)` to account for the operations saved when the empty space is part of a swap cycle. This is because if the empty space is part of the cycle, we save one swap at the start and one at the end of that cycle.
5. Call the function `f(nums, 0)` considering the empty space is at the beginning and store the result in `a`.
6. Transform the array to simulate moving the empty space to the end by decrementing each value by `1`, taking modulo `n`, and then calling the function `ff()` on this transformed array. This is stored in `b`.
7. The final answer is the minimum of `a` and `b` since sorting can be achieved with the empty space at either the start or the end, and we are asked for the minimum number of operations.

The code uses the cycle-sort algorithm concept adapted for an array with an empty space. The cycle detection and counting approach to determine the minimum number of operations make this algorithm both efficient and elegant. The pattern of simulating the empty space at different positions by transforming the array shows a clever way to adapt to the constraints of the problem without modifying the input array directly.

Example Walkthrough

Let's take a small example to illustrate the solution approach. Suppose we have an integer array `nums` given by `nums = [1, 3, 0, 2]` with `n = 4`. We want to sort this array such that the zero either ends up at the beginning `[0, 1, 2, 3]`, or at the end `[1, 2, 3, 0]`.

Here are the step-by-step operations following the solution approach:

1. We define a helper function called `f(nums, k)` that calculates the number of operations needed to sort the array with `0` at index `k`.
2. We initialize a `vis` array with the same length as `nums` to keep track of the positions we have already considered in order to avoid double counting.
3. We start iterating over the array. For the first call of `f(nums, 0)`, we will consider the empty space at the beginning:
   - Begin with `i = 0`. Since `nums[0]` is `1` which is the correct position when the empty space is considered at the beginning, `i` gets marked as visited and we continue.
   - Move to `i = 1`. The value at `nums[1]` is `3`, which is not in its correct position. A cycle is formed here: the `3` should be in `nums[3]`, and the `2` that is in `nums[3]` should be in `nums[2]`. We perform these operations moving the items into their correct positions and mark each as visited.
   - Since we did not need to use the empty space for this cycle, the number of moves are `cycles_length - 1` which equals `2 - 1 = 1`.
4. We adjust the operation count for any cycles that include the chosen position of the empty space (`k`), which is not needed here as `k` is `0`.
5. The result of `f(nums, 0)` is `1`, since we performed just one operation.
6. Now we need to transform the array to simulate the empty space being at the end. We decrement each value (except the zero) by `1` and take modulo `n` which gives us the transformed array `[0, 2, 4 % 4, 1] == [0, 2, 0, 1]`.
7. Call the function `ff()` on the transformed array `ff([0, 2, 0, 1])`. The calculation will proceed similarly but this time considering the empty space at the end of the array.
   - Begin with `i = 0`. Since `nums[0]` is `0`, it is in the correct position at the end of the array. So continue to the next index.
   - At `i = 1`, `nums[1]` is `2`, which needs to move to `nums[2]`. However, since we have two zeroes we can't proceed with standard cycle detection. But by the construction of the transformed array, we know it should have been at `nums[3]`, and hence we have an incorrect element at `nums[1]`, `nums[2]`, and `nums[3]`.
   - We count the number of operations needed to sort the wrong sequence which is `length of the sequence - 1 = 3 - 1 = 2`.
8. The result of `ff([0, 2, 0, 1])` is `1`, indicating two operations are needed to sort the array with empty space at the end.
9. The final answer is the minimum of `a` and `b`, which is `min(1, 2) = 1`.

Let's take a small example to illustrate the solution approach. Suppose we have an integer array `nums` given by `nums = [1, 3, 0, 2]` with `n = 4`. We want to sort this array such that the zero either ends up at the beginning `[0, 1, 2, 3]`, or at the end `[1, 2, 3, 0]`.

Here are the step-by-step operations following the solution approach:

1. We define a helper function called `f(nums, k)` that calculates the number of operations needed to sort the array with `0` at index `k`.
2. We initialize a `vis` array with the same length as `nums` to keep track of the positions we have already considered in order to avoid double counting.
3. We start iterating over the array. For the first call of `f(nums, 0)`, we will consider the empty space at the beginning:
   - Begin with `i = 0`. Since `nums[0]` is `1` which is the correct position when the empty space is considered at the beginning, `i` gets marked as visited and we continue.
   - Move to `i = 1`. The value at `nums[1]` is `3`, which is not in its correct position. A cycle is formed here: the `3` should be in `nums[3]`, and the `2` that is in `nums[3]` should be in `nums[2]`. We perform these operations moving the items into their correct positions and mark each as visited.
   - Since we did not need to use the empty space for this cycle, the number of moves are `cycles_length - 1` which equals `2 - 1 = 1`.
4. We adjust the operation count for any cycles that include the chosen position of the empty space (`k`), which is not needed here as `k` is `0`.
5. The result of `f(nums, 0)` is `1`, since we performed just one operation.
6. Now we need to transform the array to simulate the empty space being at the end. We decrement each value (except the zero) by `1` and take modulo `n` which gives us the transformed array `[0, 2, 4 % 4, 1] == [0, 2, 0, 1]`.
7. Call the function `ff()` on the transformed array `ff([0, 2, 0, 1])`. The calculation will proceed similarly but this time considering the empty space at the end of the array.
   - Begin with `i = 0`. Since `nums[0]` is `0`, it is in the correct position at the end of the array. So continue to the next index.
   - At `i = 1`, `nums[1]` is `2`, which needs to move to `nums[2]`. However, since we have two zeroes we can't proceed with standard cycle detection. But by the construction of the transformed array, we know it should have been at `nums[3]`, and hence we have an incorrect element at `nums[1]`, `nums[2]`, and `nums[3]`.
   - We count the number of operations needed to sort the wrong sequence which is `length of the sequence - 1 = 3 - 1 = 2`.
8. The result of `ff([0, 2, 0, 1])` is `1`, indicating two operations are needed to sort the array with empty space at the end.
9. The final answer is the minimum of `a` and `b`, which is `min(1, 2) = 1`.

Python Solution

```python
from typing import List

class Solution:
    def sortArray(self, nums: List[int]) -> int:
        # Helper function to compute the minimum swaps required to sort the array
        # while optionally adjusting the cost for a specific position k.
        def f(nums: List[int], k: int) -> int:
            n = len(nums)
            visited = [False] * n  # Initialize visited array to keep track of swapped elements.
            swap_count = 0  # Initialize swap count.

            # Iterate over the array to determine cycles and the number of swaps needed.
            for i, val in enumerate(nums):
                if i == val or visited[i]:  # Skip if already in correct position or visited.
                    continue
                j = i  # Start a new cycle.
                while not visited[j]:
                    visited[j] = True  # Mark the current element as visited.
                    swap_count += 1  # Increment swap count since we're moving this element.
                    j = nums[j]  # Move to the next element in the cycle.

            # Adjust the swap count based on the special condition when nums[k] == k.
            return swap_count - 2 * (nums[k] == k)

        # Main body of the sort_array method
        n = len(nums)

        # Compute the swaps required in the original array.
        swaps_in_original = compute_min_swaps(nums, 0)

        # Compute the swaps required in the array with elements decremented by 1 and wrapped around.
        # This adjusts positions as if the last element was at the start of the array.
        wrapped_nums = [(val - 1 + n) % n for val in nums]
        swaps_in_wrapped = compute_min_swaps(wrapped_nums, n - 1)

        # Return the minimum of the two computed swap counts.
        return min(swaps_in_original, swaps_in_wrapped)
```

Java Solution

```java
class Solution {

    public int sortArray(int[] nums) {
        int n = nums.length;
        int[] modifiedArray = new int[n];

        // Populate the modifiedArray with elements modulo n.
        for (int i = 0; i < n; ++i) {
            modifiedArray[i] = (nums[i] - 1 + n) % n;
        }

        // Compute the number of swaps needed when considering original and modified arrays.
        int originalSwaps = countSwaps(nums, 0);
        int modifiedSwaps = countSwaps(modifiedArray, n - 1);

        // Return the minimum swaps between the original and modified.
        return Math.min(originalSwaps, modifiedSwaps);
    }

    // Count the number of swaps required to sort the array such that each element i is at index i.
    private int countSwaps(int[] nums, int k) {
        boolean[] visited = new boolean[nums.length];
        int swaps = 0; // Counter for the number of swaps.

        // Scan through each element in the array.
        for (int i = 0; i < nums.length; ++i) {
            // If the element is in the right place or already visited, skip it.
            if (i == nums[i] || visited[i]) {
                continue;
            }
            // Increase swap count for the new cycle.
            swaps++;
            int j = i;
            // Follow the cycle and mark elements as visited.
            while (!visited[j]) {
                visited[j] = true;
                // Increment swap count for each visited element.
                swaps += nums[j];
                j = nums[j];
            }
        }

        // If the special element k is not in the correct position, subtract two swaps.
        if (nums[k] != k) {
            swaps -= 2;
        }

        return swaps;
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    // Method to sort the array with some custom requirements.
    int sortArray(vector<int>& nums) {
        int size = nums.size();

        // Lambda function to calculate the number of swaps required to sort,
        // considering the numbers to be positions in a cycle.
        auto countSwaps = [&](vector<int>& array, int specialIndex) {
            vector<bool> visited(size, false);
            int swapCount = 0;
            for (int i = 0; i < size; ++i) {
                // Skip if the current index corresponds to its value
                // or has already been visited during swap calculation.
                if (i == array[i] || visited[i]) continue;
                int currentIndex = i;
                // Traverse the cycle to account for the new cycle.
                ++swapCount;
                // Traverse the cycle and mark the indices as visited.
                while (!visited[currentIndex]) {
                    visited[currentIndex] = true;
                    // Increment swapCount for each swap within the cycle.
                    swapCount += array[currentIndex];
                    currentIndex = array[currentIndex];
                }
            }
            // Decrement by 2 if the specialIndex was swapped.
            if (array[specialIndex] != specialIndex) swapCount -= 2;
            return swapCount;
        };

        // Calculate the number of swaps normally.
        int normalSwaps = countSwaps(nums, 0);

        // Create a copy of the original array to try swapping with an offset.
        vector<int> adjustedArray = nums;

        // Adjust each number in the array by decreasing 1, then modulo by size.
        for (int i = 0; i < adjustedArray.size(); ++i) {
            adjustedArray[i] = (adjustedArray[i] - 1 + size) % size;
        }

        // Calculate the number of swaps with the special (last) position.
        int adjustedSwaps = countSwaps(adjustedArray, size - 1);

        // Return the minimum number of swaps between the two approaches.
        return min(normalSwaps, adjustedSwaps);
    }
};
```

Typescript Solution

```typescript
// Imported required JavaScript functions
import { min } from 'lodash';

// Function to sort the array with some custom requirements.
function sortArray(nums: number[]): number {
    const size: number = nums.length;

    // Lambda function to calculate the number of swaps required to sort,
    // considering the numbers to be positions in a cycle.
    const countSwaps = (array: number[], specialIndex: number): number => {
        const visited: boolean[] = new Array(size).fill(false);
        let swapCount = 0;
        for (let i = 0; i < size; ++i) {
            // Skip if the current index corresponds to its value
            // or has already been visited during swap calculation.
            if (i === array[i] || visited[i]) continue;
            let currentIndex = i;
            // Increment swapCount for each new cycle.
            ++swapCount;
            // Traverse the cycle and mark the indices as visited.
            while (!visited[currentIndex]) {
                visited[currentIndex] = true;
                // Increment swapCount for each swap within the cycle.
                swapCount += array[currentIndex];
                currentIndex = array[currentIndex];
            }
        }
        // Decrement by 2 if the specialIndex was swapped.
        if (array[specialIndex] !== specialIndex) swapCount -= 2;
        return swapCount;
    };

    // Calculate the number of swaps normally.
    const normalSwaps = countSwaps(nums, 0);

    // Create a copy of the original array to try swapping with an offset.
    const adjustedArray: number[] = nums.slice();

    // Adjust each number in the array by decreasing 1, then modulo by size.
    for (let i = 0; i < adjustedArray.length; ++i) {
        adjustedArray[i] = (adjustedArray[i] - 1 + size) % size;
    }

    // Calculate the number of swaps with the special (last) position.
    const adjustedSwaps = countSwaps(adjustedArray, size - 1);

    // Return the minimum number of swaps between the two approaches.
    return min([normalSwaps, adjustedSwaps]);
}

// Example of calling the function with an array
const exampleArray = [3, 2, 1, 0];
const minSwapsRequired = sortArray(exampleArray);
console.log('Minimum number of swaps required: ${minSwapsRequired}');
```

Time and Space Complexity

Time Complexity

The given code consists mainly of two parts for calculating time complexity:

- A helper function `f` which traverses each element to create cycles and calculate the number of swaps needed to sort the array.
- Two invocations of the `f` function with different arguments.

In the `f` function, we iterate through the array of length `n` exactly once in the outer loop. Within the loop, every index in the array is visited at most once due to the presence of the `vis` array which keeps the track of visited indices to prevent revisiting. Because each index is visited exactly once overall (sum of outer and inner while loop), the complexity of this part is $O(n)$.

The helper function is called twice with different arguments, but the time complexity of each call remains $O(n)$, thus not changing the overall time complexity.

Therefore, the total time complexity of the given code is $O(n)$.

Space Complexity

As for the space complexity, we are using:

- A boolean visited array `vis` of size `n`.
- Two integer variables `cnt` and `j`.

Hence, the space complexity is dominated by the size of the `vis` array, which is $O(n)$.

In conclusion, the time complexity of the given code is $O(n)$, and the space complexity is $O(n)$.