912. Sort an Array

Medium Array Divide and Conquer Bucket Sort Counting Sort Radix Sort Sorting Heap (Priority Queue) Merge Sort
Leetcode Link

Problem Description

The problem requires us to sort a given array of integers, called `nums`, in ascending order. The challenge is to achieve this without using any of the sorting functions built into the programming libraries. Additionally, we are tasked with sorting the array within a time complexity of `O(nlog(n))`, which is typically the time complexity of efficient sorting algorithms like quicksort, mergesort, and heapsort. Moreover, we need to aim for the smallest space complexity possible, which suggests we should use an in-place sorting algorithm that doesn't require allocating additional space proportional to its input size.

Intuition

To meet the `O(nlog(n))` time complexity requirement without using built-in functions, we can use a divide and conquer algorithm like quicksort. Quicksort works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. The intuition behind quicksort is that by dealing with smaller sub-arrays, we can efficiently organize the elements in a divide and conquer manner.

The provided solution employs the quicksort algorithm in its in-place version to achieve the smallest space complexity possible. Here's the thought process:

1. A random pivot element `x` from the array is chosen to avoid the worst-case performance of quicksort on already sorted or reverse-sorted data, which would lead to `O(n^2)` time complexity.
2. We perform the partitioning step by using three pointers: `i` is the end of the partition where all elements are less than `x`, `j` is the beginning of the partition where all elements are greater than `x`, and `k` iterates over each element to compare them with the pivot.
3. During partitioning, if an element is less than `x`, we swap it with the first element of the greater-than-pivot partition, effectively growing the less-than-pivot partition by one. If an element is greater than `x`, we swap it with the element just before the beginning of the greater partition. When an element is equal to `x`, `k` is simply advanced by one.
4. Once the partitioning is done, recursively apply the same process to the sub-arrays formed on either side of the pivot's final position (before `i` and after `j`).
5. The process is repeated until the base case is reached (sub-array with zero or one element), at which point the sub-array is considered sorted.

By recursively sorting sub-arrays, and not creating additional arrays in the process, the provided quicksort algorithm sorts the input array in-place with `O(nlog(n))` average time complexity and `O(log(n))` space complexity due to the stack space used by the recursion.

Solution Approach

The solution provided implements an in-place quicksort algorithm to sort the array. Let's walk through the key steps and patterns used, referencing the code:

1. The main function `sortArray` contains a nested function `quick_sort`, which is a common pattern to encapsulate the recursive logic within the sorting function and allows us to use variables from the outer scope.
2. `quick_sort` takes two arguments, `l` and `r`, which are the indices of the left and right boundaries of the sub-array that it needs to sort.
3. The termination condition for the recursion is when the left boundary `l` is greater than or equal to the right boundary `r`. At this point, the sub-array has zero or one element and is considered sorted.
4. A pivot element `x` is chosen using `randint(l, r)` to randomly select an index between `l` and `r`, inclusive. The random pivot mitigates the risk of encountering the worst-case time complexity.
5. Three pointers are established: `i` is initially set to one less than the left boundary `l`, `j` is one more than the right boundary `r`, and `k` is set to the left boundary `l`.
6. An iterative process starts where the `k` pointer moves from `l` to `j`. During this process:
   - If the current element `nums[k]` is less than the pivot `x`, it is swapped with the element at `i+1`, and both `i` and `k` are incremented. This effectively moves the current element to the left partition (elements less than the pivot).
   - If `nums[k]` is greater than the pivot `x`, the current element is swapped with the element just before `j`, and `j` is decreased. This moves the current element to the right partition (elements greater than the pivot).
   - If `nums[k]` is equal to the pivot `x`, only `k` is incremented since the element at `k` is already equal to the pivot, and we want to keep this partition in the middle.
7. The array now has three partitions: elements less than the pivot (`l` to `i`), elements equal to the pivot (`i+1` to `j-1`), and elements greater than the pivot (`j` to `r`). The elements equal to the pivot are already at their final destination.
8. Recursive calls are made to `quick_sort` for the left partition (`l` to `i`) and the right partition (`j` to `r`). Note that the elements equal to the pivot are not included.
9. Once all recursive calls are completed, the entire array has been sorted in place, and the sorted array `nums` is returned.

Data Structures Used:

- The primary data structure is the input array `nums`. No additional data structure is utilized, which contributes to the algorithm's small space complexity.

Algorithms and Patterns:

- The solution is a classic example of the divide and conquer algorithm (quicksort) and demonstrates the use of recursion and in-place array manipulation.
- Random pivot selection is used to improve the expected performance by reducing the likelihood of pathological cases that lead to poor performance.

Overall, this approach respects the problem constraints by avoiding built-in functions and aiming for optimal time and space complexities.

Example Walkthrough

Let's walk through an example to illustrate the solution approach using the quicksort algorithm, with the array `[5, 3, 8, 4, 2, 7, 1, 6]`.

1. Call `sortArray` on the entire array.
2. Choose a random pivot, say the value at index 0 (in this case, `5`). Set up our pointers: `i` to -1 (one less than the start index 0), `j` to 8 (one more than the end index 7), and `k` to 0.
3. Start the partitioning process by iterating `k` from 0 to `j`. We encounter the following cases:
   - At `k=0`, `nums[k]` is equal to the pivot `5`, `k` is incremented to 1.
   - At `k=1`, `nums[k]` is `3`, which is less than `5`. Swap `nums[k]` with `nums[i+1] (3 with 5)`, increment `i` and `k`.
   - At `k=2`, `nums[k]` is `8`, which is greater than `5`. Swap `nums[k]` with `nums[j-1] (8 with 6)`, decrement `j`.
   - At `k=2` again (since `k` doesn't move), `nums[k]` is now `6`, which is greater than `5`. Swap `nums[k]` with `nums[j-1]`, decrement `j`.
   - Continue this process until all elements are partitioned around the pivot (`k` reaches the minimum `j`).
4. The array is now partitioned into `[3, 2, 1, 5, 5, 7, 6, 8]`.
5. Recursively apply `quick_sort` to the sub-array less than the pivot `[3, 2, 1]`, and to the sub-array greater than the pivot `[7, 6, 8]`.
6. For the left sub-array `[3, 2, 1]`:
   - Choose pivot, let's say `3`.
   - Partition around `3`. The sub-array becomes `[2, 1, 3]`.
   - Recursively sort `[2, 1]`. Choose pivot `2`. Partition around `2`. The sub-array becomes `[1, 2]`.
   - No more sorting necessary as each sub-array is of length 1 or in proper order.
7. Repeat this process for the right sub-array with `7, 6, 8`.
8. After sorting both sub-arrays and considering that elements equal to pivot are in the correct place, the initial array `[5, 3, 8, 4, 2, 7, 1, 6]` becomes `[1, 2, 3, 5, 5, 6, 7, 8]`.

By continuing this recursive partitioning and sorting, the final sorted array is obtained.

Python Solution

```python
from random import randint
from typing import List

class Solution:
    def sortArray(self, nums: List[int]) -> List[int]:
        # Helper function to perform quick sort.
        def quick_sort(left, right):
            # Base case: If the current segment is empty or has only one element, no need to sort.
            if left >= right:
                return

            # Choose a random pivot element from the segment.
            pivot = nums[randint(left, right)]
            # Initialize pointers:
            # 'less_than_pointer' marks the end of the segment with elements less than pivot.
            # 'greater_than_pointer' marks the start of the segment with elements greater than pivot.
            # 'current' is used to scan through the list.
            less_than_pointer, greater_than_pointer, current = left - 1, right + 1, left

            # Iterate until 'current' is less than 'greater_than_pointer'.
            while current < greater_than_pointer:
                if nums[current] < pivot:
                    # Move the element to the segment with elements less than pivot.
                    less_than_pointer += 1
                    nums[less_than_pointer], nums[current] = nums[current], nums[less_than_pointer]
                    current += 1
                elif nums[current] > pivot:
                    # Move the element to the segment with elements greater than pivot.
                    greater_than_pointer -= 1
                    nums[greater_than_pointer], nums[current] = nums[current], nums[greater_than_pointer]
                else:
                    # If the current element is equal to the pivot, move to the next element.
                    current += 1

            # Recursively apply quick sort to the segment with elements less than pivot.
            quick_sort(left, less_than_pointer)
            # Recursively apply quick sort to the segment with elements greater than pivot.
            quick_sort(greater_than_pointer, right)

        # Call the quick_sort function with the initial boundaries of the list.
        quick_sort(0, len(nums) - 1)
        # Return the sorted list.
        return nums
```

Java Solution

```java
class Solution {
    private int[] nums; // this array holds the numbers to be sorted

    public int[] sortArray(int[] nums) {
        this.nums = nums; // initialize the nums array with the input array
        quickSort(0, nums.length - 1); // call the quickSort method on the entire array
        return nums; // return the sorted array
    }

    private void quickSort(int left, int right) {
        if (left >= right) { // base case for recursion (if the subarray has 1 or no element)
            return;
        }
        int pivot = nums[(left + right) >> 1]; // choose the middle element as the pivot
        int i = left - 1, j = right + 1; // initialize pointers for the two subarrays
        while (i < j) { // continue until the pointers cross
            while (nums[++i] < pivot) { // increment i until an element larger than the pivot is found
            }
            while (nums[--j] > pivot) { // decrement j until an element smaller than the pivot is found
            }
            if (i < j) { // if the pointers haven't crossed, swap the elements
                int temp = nums[i];
                nums[i] = nums[j];
                nums[j] = temp;
            }
        }
        quickSort(left, j); // recursively apply quickSort to the subarray to the left of the pivot
        quickSort(j + 1, right); // recursively apply quickSort to the subarray to the right of the pivot
    }
}
```

C++ Solution

```cpp
#include <vector>
#include <functional> // For std::function

class Solution {
public:
    vector<int> sortArray(vector<int>& nums) {
        // A lambda function for recursive quick sort
        std::function<void(int, int)> quickSort = [&](int left, int right) {
            // Base case: If the current segment is empty or a single element, no need to sort
            if (left >= right) {
                return;
            }

            // Initialize pointers for partitioning process
            int pivotIndex = left + (right - left) / 2; // Choose middle element as pivot
            int pivotValue = nums[pivotIndex];
            int i = left - 1;
            int j = right + 1;

            // Partition the array into two halves
            while (i < j) {
                // Increment i until nums[i] is greater or equal to pivot
                do {
                    i++;
                } while (nums[i] < pivotValue);

                // Decrement j until nums[j] is less or equal to pivot
                do {
                    j--;
                } while (nums[j] > pivotValue);

                // If i and j haven't crossed each other, swap the elements
                if (i < j) {
                    std::swap(nums[i], nums[j]);
                }
            }

            // Recursively apply the same logic to the left and right halves of the array
            quickSort(left, j);     // Apply quicksort to the left subarray
            quickSort(j + 1, right); // Apply quicksort to the right subarray
        };

        // Start the quick sort from the first to the last element
        quickSort(0, nums.size() - 1);

        // Return the sorted array
        return nums;
    }
};
```

Typescript Solution

```typescript
// This function sorts an array of numbers using Quick Sort algorithm.
function sortArray(nums: number[]): number[] {
    // Helper function to perform the quickSort algorithm.
    function quickSort(left: number, right: number) {
        // If the current segment is empty or has one element, no sorting is needed.
        if (left >= right) {
            return;
        }

        let i = left - 1;
        let j = right + 1;

        // Choose the pivot element from the middle of the segment.
        const pivot = nums[(left + right) >> 1];

        // Partition process: elements < pivot go to the left, elements > pivot go to the right.
        while (i < j) {
            // Find left element greater than or equal to the pivot.
            while (nums[++i] < pivot);
            // Find right element less than or equal to the pivot.
            while (nums[--j] > pivot);

            // If pointers have not crossed, swap the elements.
            if (i < j) {
                [nums[i], nums[j]] = [nums[j], nums[i]];
            }
        }

        // Recursively apply the same logic to the left partition.
        quickSort(left, j);
        // Recursively apply the same logic to the right partition.
        quickSort(j + 1, right);
    }

    // Obtain the length of the array to sort.
    const n = nums.length;
    // Call the quickSort helper function on the entire array.
    quickSort(0, n - 1);

    // Return the sorted array.
    return nums;
}
```

Time and Space Complexity

Time Complexity

The given Python code implements the Quick Sort algorithm with a three-way partitioning approach. Let's break down the time complexity:

- The `quick_sort` function is recursively called on subarrays of the initial array `nums`. In the average case, where the pivot selected by `randint(l, r)` happens to divide the array into relatively equal parts, each level of recursion deals with half the size of the previous level, resulting in `O(log n)` levels (with `n` being the number of elements in `nums`).
- On each level of recursion, the algorithm iterates through the current subarray once, partitioning it into elements less than, equal to, and greater than the pivot. This partitioning takes `O(n)` time at each level.
- Combining these two observations, the average-case time complexity is `O(n log n)`.

However, in the worst case, when the pivot is always the smallest or the largest element after partitioning, the recursion depth becomes `O(n)`, with each level taking linear time to partition. Therefore, the worst-case time complexity is `O(n^2)`.

Space Complexity

As for space complexity, since the Quick Sort implementation provided is in-place (it doesn't create additional arrays for partitioning):

- The primary space usage comes from the call stack due to recursion. In the average case, the maximum depth of the call stack will be `O(log n)`, hence the space complexity is `O(log n)`.
- In the worst case, where the array is partitioned into a single element and the rest at every step, the maximum depth of the recursion could be `O(n)`. Therefore, the worst-case space complexity would also be `O(n)`.

Overall, the Quick Sort provided performs well on average but has a worse time complexity in the worst-case scenario. Its space complexity is relatively low, being logarithmic in the average case, and only gets higher when the pivot choices consistently result in unbalanced partitions.