4. Median of Two Sorted Arrays

Hard Array Binary Search Divide and Conquer Leetcode Link

Problem Description

The problem presents us with two sorted arrays, `nums1` and `nums2`, with sizes `m` and `n` respectively. Our task is to find the median of these two sorted arrays combined. The median is the value that separates a dataset into two halves. In other words, half of the numbers in the dataset are less than the median, and half are greater than it. If there is an odd number of observations, the median is the middle number. If there is an even number of observations, the median is the average of the two middle numbers.

Given the arrays are already sorted, if we could merge them, finding the median would be trivial: take the middle value or the average of the two middle values. However, merging the arrays may not be efficient, specifically for large sizes of `m` and `n`. Therefore, we are asked to find an efficient approach with a runtime complexity of O(log (m+n)), which suggests that we should not actually merge the arrays, and instead, consider an algorithm similar in efficiency to binary search.

Intuition

The intuition behind the solution is to use a binary search due to the sorted nature of the arrays and the requirement of a O(log (m+n)) runtime complexity. To find the median of two sorted arrays, we could approach the problem as finding the 'kth' smallest element, where 'k' would be the median position in the combined array, which is `(m+n+1)//2` for the lower median and `(m+n+2)//2` for the upper median.

The solution uses a binary search which is applied recursively by defining a helper function `f(i, j, k)` where `i` and `j` are the current indices in `nums1` and `nums2` respectively, and `k` is the position of the element we are looking for in the merged array. If one of the arrays is exhausted (`i >= m` or `j >= n`), the kth element is directly found in the other array. When the position `k` is 1, we simply take the minimum between the current elements pointed by `i` and `j`.

The main trick is dividing the problem in half at each step: We pick the middle element position (`p = k // 2`) from the remaining elements to compare in each array and reduce the problem size by `p`, discarding the smaller half each time. The variable `x` corresponds to the element at position `i + p - 1` in `nums1` unless this position is outside the array bounds (in which case we use infinity), and similarly for `y` in `nums2`. We recursively call function `f` with adjusted indices based on which half we are discarding.

Finally, since the median can be a single middle value (for odd total length) or the average of two middle values (for even total length), the solution calls the helper function twice, for the kth and (k+1)th positions, and returns their average.

Note: The use of `inf` in the solution assumes a Python environment where `inf` represents an infinite number, which serves as a stand-in when comparing out-of-bounds indices in one of the arrays.

Solution Approach

The solution employs a divide and conquer strategy which is effectively a tailored binary search across two arrays to find the median.

Here's a step by step breakdown of the approach:

1. **Define a Helper Function:** The `f(i, j, k)` function serves as the core for the binary search strategy, by recursively finding the `kth` smallest element in the combined array made up by `nums1` and `nums2`. Parameters `i` and `j` are the starting points in `nums1` and `nums2`, respectively, and `k` is the number of steps away from the starting points to the target element in the merged array.
2. **Base Cases:** The helper function has three essential base cases:
   - If `i` or `j` run past the length of their respective arrays (`i >= m` or `j >= n`), the answer is found in the other array. This is because once one array is exhausted, the remaining kth element must be in the other array.
   - If `k` is 1, we can't divide the search space any smaller, so we just return the minimum of the two element values that `i` and `j` are pointing at.
3. **Recursive Case:** For the recursive case, we divide the remaining search space (`k`) in two by setting `p = k // 2`. We then find the potential `pth` element in both arrays (`x` and `y`) if those positions are in bounds.
4. **Compare and Eliminate:** We compare these potential candidates, `x` and `y`, to eliminate the half of the search space that can't possibly contain the kth smallest element. If `x` is smaller, we know that anything before and including `x` in `nums1` cannot contain the kth smallest, so we move the search space past `x`, by calling `f(i + p, j, k - p)`. Similarly, if `y` is smaller, we call `f(i, j + p, k - p)`.
5. **Find Medians:** Once we've established our helper function, we use it to find the medians depending on whether the combined length of `nums1` and `nums2` is odd or even:
   - For an odd combined length, the median is at position `(m + n + 1) // 2`, so only one call to `f(0, 0, (m + n + 1) // 2)` is necessary.
   - For an even combined length, we call the function twice with `(m + n + 1) // 2` and `(m + n + 2) // 2` to derive two middle values and take their average.

The use of "inf" ensures that the comparison logic remains correct even when all elements of one array have been accounted for, thus allowing the binary search to proceed correctly with the remaining array.

Data Structures:

- No additional data structures are used, aside from the input arrays and variables to store intermediate values.

Algorithm Patterns:

- Binary Search: The solution utilizes the binary search pattern in a recursive manner to find the kth smallest element in a merged sorted array without actually merging them.
- Divide and Conquer: The problem space is halved in each recursive call, efficiently narrowing down to the median value.
- Recursion: Through the recursive function `f(i, j, k)`, we are capable of diving into smaller subproblems and compositions of the problem.

Overall, the binary search pattern is applied in a unique way that leverages the sorted property of the arrays and the fact that the kth smallest element search can be performed in a log-linear time complexity given these properties.

Example Walkthrough

Let's go through a small example to illustrate the solution approach using two sorted arrays `nums1` and `nums2`.

Consider the arrays `nums1 = [1, 3]` and `nums2 = [2]`. The combined array after merging would be `[1, 2, 3]`, where the median is `2` because it is the middle element.

We will not actually merge the arrays. Instead, we will employ the divide and conquer binary search as described.

Step 1: Find the total length and calculate the median position.

The total length of the combined arrays is `m + n = 2 + 1 = 3`, which is odd. The median position `k` is `(m + n + 1) // 2 = (3 + 1) // 2 = 2`.

Step 2: Begin the binary search with the helper function `f(i, j, k)`.

We want to find the 2nd smallest element without merging, so we start by calling `f(0, 0, 2)`. Here `i` and `j` are starting indices for `nums1` and `nums2`, and `k` is 2.

Step 3: Handle the base and recursive cases.

Since `k` is not 1 and we haven't exceeded array lengths, we find the middle index `p` for the current `k`, which is `p = k // 2 = 1`.

Now, we look at the elements at index `p - 1` (since arrays are zero-indexed) in each array. In `nums1`, it's `nums1[0]`, which is `1`, and in `nums2`, it's `nums2[0]`, which is `2`. (We will call these `x` and `y` respectively).

Step 4: Eliminate one half of the search space.

Since `x < y` (1 < 2), we eliminate `x` and elements to the left of `x`. Now we call `f(i + p, j, k - p)`, which translates to `f(1, 0, 1)` because we're moving past the element `1` in `nums1`.

Step 5: Reaching the base case.

Now `k` is `1`, which means we just need to find the minimum of the current elements `nums1[i]` and `nums2[j]`. The array `nums1` at index `1` has no more elements, and the array `nums2` at index `0` has the element `2`.

Since the element in `nums1` is out of bounds, we consider it infinity. Therefore, `nums2[j]` which is `2`, is the smaller element.

Step 6: Return the result.

Since we were looking for the 2nd smallest element, `2` is the median, and we return this as the result.

Through this example, we successfully applied the binary search divide and conquer strategy to find the median of two sorted arrays without merging them, fulfilling the required O(log (m+n)) runtime complexity.

Python Solution

```python
from typing import List

class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        # Helper function to find the k-th smallest element in two sorted arrays
        def findKthElement(index1: int, index2: int, k: int) -> int:
            # If nums1 is exhausted, return k-th element from nums2
            if index1 >= length1:
                return nums2[index2 + k - 1]
            # If nums2 is exhausted, return k-th element from nums1
            if index2 >= length2:
                return nums1[index1 + k - 1]
            # If k is 1, return the smaller of the two starting elements
            if k == 1:
                return min(nums1[index1], nums2[index2])

            # Compare elements at half of k in each array and call recursively
            half_k = k // 2
            midVal1 = nums1[index1 + half_k - 1] if index1 + half_k - 1 < length1 else float('inf')
            midVal2 = nums2[index2 + half_k - 1] if index2 + half_k - 1 < length2 else float('inf')

            # Recurse into the array with the smaller value or adjust indexes accordingly
            if midVal1 < midVal2:
                return findKthElement(index1 + half_k, index2, k - half_k)
            else:
                return findKthElement(index1, index2 + half_k, k - half_k)

        # Get lengths of both arrays
        length1, length2 = len(nums1), len(nums2)

        # Find the median
        # If the combined length is odd, the median is the middle element; if even, the average of the two middle elements
        median_left = findKthElement(0, 0, (length1 + length2 + 1) // 2)
        median_right = findKthElement(0, 0, (length1 + length2 + 2) // 2)

        # Calculate the median and return it
        return (median_left + median_right) / 2

```

Java Solution

```java
class Solution {
    // Class variables to store the size of arrays and the arrays themselves.
    private int sizeNums1;
    private int sizeNums2;
    private int[] nums1;
    private int[] nums2;

    // Main function to find the median of two sorted arrays.
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        // Initialize class variables with array sizes and the arrays themselves.
        sizeNums1 = nums1.length;
        sizeNums2 = nums2.length;
        this.nums1 = nums1;
        this.nums2 = nums2;

        // Median can be the average of the middle two values for even length combined arrays
        // or the middle one for odd length combined arrays.
        int leftMedian = findKthElement(0, 0, (sizeNums1 + sizeNums2 + 1) / 2);
        int rightMedian = findKthElement(0, 0, (sizeNums1 + sizeNums2 + 2) / 2);

        // The median is the average of the two middle numbers for even-sized arrays.
        return (leftMedian + rightMedian) / 2.0;
    }

    // Helper function to find the k-th element.
    private int findKthElement(int startNums1, int startNums2, int k) {
        // Base cases for recursion.
        if (startNums1 >= sizeNums1) {
            return nums2[startNums2 + k - 1]; // Select k-th element from nums2
        }
        if (startNums2 >= sizeNums2) {
            return nums1[startNums1 + k - 1]; // Select k-th element from nums1
        }
        if (k == 1) {
            // If k is 1, then return the minimum of current starting elements.
            return Math.min(nums1[startNums1], nums2[startNums2]);
        }

        // Calculate the mid point to compare elements.
        int midIndex = k / 2;
        // Assign INT_MAX if the mid point is beyond the array bounds.
        int midValNums1 = startNums1 + midIndex - 1 < sizeNums1 ? nums1[startNums1 + midIndex - 1] : Integer.MAX_VALUE;
        int midValNums2 = startNums2 + midIndex - 1 < sizeNums2 ? nums2[startNums2 + midIndex - 1] : Integer.MAX_VALUE;

        // Discard half of the elements from the array which has smaller midValue.
        // Because those can never contain the k-th element.
        if (midValNums1 < midValNums2) {
            return findKthElement(startNums1 + midIndex, startNums2, k - midIndex);
        } else {
            return findKthElement(startNums1, startNums2 + midIndex, k - midIndex);
        }
    }
}

```

C++ Solution

```cpp
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        // Calculate the lengths of both input arrays
        int length1 = nums1.size();
        int length2 = nums2.size();

        // Define a lambda function to find the k-th element in the merged array
        auto findKthElement = [&](int start1, int start2, int k) -> int {
            // If start1 is beyond the end of nums1, k-th element is in nums2
            if (start1 >= length1) {
                return nums2[start2 + k - 1];
            }
            // If start2 is beyond the end of nums2, k-th element is in nums1
            if (start2 >= length2) {
                return nums1[start1 + k - 1];
            }
            // If k is 1, return the minimum of the first unchecked element in both arrays
            if (k == 1) {
                return std::min(nums1[start1], nums2[start2]);
            }
            // Find the midpoint of the remaining k elements to compare
            int half_k = k / 2;

            // Check the boundary conditions and select the value to compare
            int value1 = start1 + half_k - 1 < length1 ? nums1[start1 + half_k - 1] : INT_MAX;
            int value2 = start2 + half_k - 1 < length2 ? nums2[start2 + half_k - 1] : INT_MAX;

            // If value1 is less than value2, the k-th element is not in the first half_k elements of nums1
            // Otherwise, it is not in the first half_k elements of nums2
            if (value1 < value2) {
                return findKthElement(start1 + half_k, start2, k - half_k);
            } else {
                return findKthElement(start1, start2 + half_k, k - half_k);
            }
        };

        // Find the median by averaging the middle two elements
        // If the total length is odd, both a and b will be the same element (the true median)
        int a = findKthElement(0, 0, (length1 + length2 + 1) / 2);
        int b = findKthElement(0, 0, (length1 + length2 + 2) / 2);

        // Calculate the final median and return it
        return (a + b) / 2.0;
    }
};

```

Typescript Solution

```typescript
function findMedianSortedArrays(nums1: number[], nums2: number[]): number {
    // Calculate the length of both input arrays
    const lengthNums1 = nums1.length;
    const lengthNums2 = nums2.length;

    // Helper function to find the kth smallest element in the two sorted arrays
    // starting from indices `indexNums1` and `indexNums2`.
    const findKthElement = (indexNums1: number, indexNums2: number, k: number): number => {
        // If the starting index for nums1 is out of range, return kth element from nums2
        if (indexNums1 >= lengthNums1) {
            return nums2[indexNums2 + k - 1];
        }
        // If the starting index for nums2 is out of range, return kth element from nums1
        if (indexNums2 >= lengthNums2) {
            return nums1[indexNums1 + k - 1];
        }
        // If k is 1, return the minimum of the first element of both arrays
        if (k === 1) {
            return Math.min(nums1[indexNums1], nums2[indexNums2]);
        }

        // Determine the k/2 element in each array or set to a very large number if it's out of range
        const halfK = Math.floor(k / 2);
        const nums1Key = indexNums1 + halfK - 1 < lengthNums1 ? nums1[indexNums1 + halfK - 1] : Number.MAX_VALUE;
        const nums2Key = indexNums2 + halfK - 1 < lengthNums2 ? nums2[indexNums2 + halfK - 1] : Number.MAX_VALUE;

        // Recur on the half of the arrays where the kth element is likely to be
        return nums1Key < nums2Key
            ? findKthElement(indexNums1 + halfK, indexNums2, k - halfK)
            : findKthElement(indexNums1, indexNums2 + halfK, k - halfK);
    };

    // Calculate indices for median elements
    const totalLength = lengthNums1 + lengthNums2;
    const medianLeftIndex = Math.floor((totalLength + 1) / 2);
    const medianRightIndex = Math.floor((totalLength + 2) / 2);

    // Find the two median elements
    const medianLeft = findKthElement(0, 0, medianLeftIndex);
    const medianRight = findKthElement(0, 0, medianRightIndex);

    // Calculate and return the median
    // If the total length of the combined array is odd, medianLeft and medianRight will be the same
    return (medianLeft + medianRight) / 2;
}

```

Time and Space Complexity

The given Python code implements the solution to find the median of two sorted arrays using a divide and conquer approach. Let's analyze the time and space complexity of the code:

Time Complexity:

The time complexity of the function can be understood by considering the primary operation it performs - finding the k-th smallest element in two sorted arrays `nums1` and `nums2`. Each recursive call to the function `f(i, j, k)` reduces the `k` by approximately half (`k - p`, where `p = k // 2`), and at the same time, it incrementally discards `p` elements from one of the arrays by advancing the index `i` or `j`.

Therefore, the `k` reduces to `1` after approximately `log(k)` steps. Since the initial `k` is derived from the total number of elements in both arrays (`(m + n + 1) // 2` or `(m + n + 2) // 2`), the overall time complexity can be estimated as `O(log(m + n))`.

Space Complexity:

There is no additional space allocated for storing elements. The space complexity is determined by the depth of the recursion stack. Since we have approximately `log(m + n)` recursive calls due to the divide and conquer algorithm, and considering each recursive call uses a constant amount of space, the space complexity is `O(log(m + n))`.