****

**SIMATS SCHOOL OF ENGINEERING**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL**

**SCIENCES, CHENNAI – 602 105**

**BONAFIDE CERTIFICATE**

**Certified that is Capstone project report “****Median of Two Sorted Arrays Using Divide and Conquer Method “is the**

**Bonafide work of “N Radhika” (192210126) who carried out the Capstone project work under my supervision**

**DR.**

**Dr. R Dhanalakshmi**

**COURSE FACULTY**

**Professor**

**Department of**

**SIMATS Engineering**

**Saveetha Institute of Medical and**

**Technical Sciences**

**Chennai – 602 105**

**EXAMINER SIGNATURE**

**HEAD OF DEPARTMENT**

**Professor**

**Department of**

**SIMATS Engineering**

**Saveetha Institute of Medical and**

**Technical Sciences**

**Chennai – 602 105**

**EXAMINER SIGNATURE**

****

**“Median of Two Sorted Arrays Using Divide And Conquer Method”**

**A Project report**

**CSA0656- Design and Analysis of Algorithms for Asymptotic Notations**

**Submitted to**

**SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES**

**In partial fulfilment for the award of the**

**degree of**

**BACHELOR OF TECHNOLOGY IN**

**ARTIFICAL INTELLIGENCE AND MACHINE LEARNING**

**by**

**N Radhika,192210126**

**Supervisor**

**Dr .R.Dhanalakshmi**

**July 2024.**

**ABSTRACT**

The problem of finding the median of two sorted arrays involves determining the middle value of a combined sorted array formed by merging two given sorted arrays, nums1 and nums2. Given the constraint that the overall runtime complexity should be O (log(m + n)), where m and n are the sizes of the two arrays, the challenge is to find an efficient approach that avoids the O (m +n) time complexity associated with merging the arrays.

The optimal solution utilizes a binary search method on the smaller of the two arrays to partition both arrays into two halves such that:

* All elements in the left half of the combined array are less than or equal to all elements in the right half.
* The median can be computed from the maximum value of the left half and the minimum value of the right half.

This approach ensures that the solution operates within the desired time complexity of O (log(m +n)) by efficiently narrowing down the correct partition point through binary search, rather than explicitly merging the arrays. The method handles edge cases such as arrays of different lengths and ensures correctness by validating partition conditions at each step.

**INTRODUCTION**

Finding the median of two sorted arrays is a well-known problem in computer science, especially useful for understanding advanced algorithms and data structures. The challenge is to determine the median value from two sorted arrays without fully merging them, while ensuring the time complexity remains efficient.

**Median Definition:**

* **Odd Length:** If the total number of elements in the combined arrays is odd, the median is the middle element.
* **Even Length:** If the total number of elements is even, the median is the average of the two middle elements.

**Problem Statement:** Given two sorted arrays nums1 and nums2, of sizes m and n respectively, return the median of the combined sorted array. The challenge is to do this in O(log(m+n)) time complexity.

Example Walkthrough:

For the arrays:

* nums1 = [1, 3]
* nums2 = [2]

Step-by-Step Explanation:

1. Combine Arrays Conceptually:
   * If we merged these two arrays, we get: [1, 2, 3]
   * The median of [1, 2, 3] is 2, as it is the middle element.
2. Efficient Approach Without Full Merge:
   * Binary Search Technique: Instead of merging, use binary search on the smaller array to find the correct partition point such that all elements on the left side of the partition are less than or equal to all elements on the right side.
   * **Partition Concept:**
     + Partition nums1 into two halves: left1 and right1.
     + Partition nums2 into two halves: left2 and right2.
     + Ensure that every element in left1 and left2 is less than or equal to every element in right1 and right2.
3. **Binary Search Logic:**
   * Set up partitions**:** Calculate the partition indices for both arrays.
   * Compare boundary values: Check if the maximum values in the left partitions are less than or equal to the minimum values in the right partitions.
   * Adjust partitions: Move the partition indices based on the comparisons until the correct partition is found.
4. Determine Median:
   * Odd Combined Length**:** Return the maximum value from the left partitions.
   * Even Combined Length: Return the average of the maximum value from the left partitions and the minimum value from the right partitions.

**Why O s(log(m + n)) Complexity?**

* Binary Search Efficiency**:** By applying binary search to the smaller of the two arrays, the algorithm efficiently narrows down the possible partition points in logarithmic time relative to the size of the smaller array.
* Avoid Full Merge: This approach avoids the linear time complexity of merging the arrays and directly calculates the median, making it suitable for large datasets.

This approach ensures that even for large arrays, the median can be found efficiently, making it a crucial technique in scenarios where performance and time complexity are important.

**CODE**

#include <stdio.h>

double findMedianSortedArrays(int\* nums1, int nums1Size, int\* nums2, int nums2Size) {

    if (nums1Size > nums2Size) {

        int\* temp = nums1;

        nums1 = nums2;

        nums2 = temp;

        int tmp = nums1Size;

        nums1Size = nums2Size;

        nums2Size = tmp;

    }

    int imin = 0, imax = nums1Size, halfLen = (nums1Size + nums2Size + 1) / 2;

    while (imin <= imax) {

        int i = (imin + imax) / 2;

        int j = halfLen - i;

        if (i < nums1Size && nums1[i] < nums2[j - 1]) {

            imin = i + 1;

        } else if (i > 0 && nums1[i - 1] > nums2[j]) {

            imax = i - 1;

        } else {

            int maxLeft;

            if (i == 0) {

                maxLeft = nums2[j - 1];

            } else if (j == 0) {

                maxLeft = nums1[i - 1];

            } else {

                maxLeft = (nums1[i - 1] > nums2[j - 1]) ? nums1[i - 1] : nums2[j - 1];

            }

            if ((nums1Size + nums2Size) % 2 == 1) {

                return maxLeft;

            }

            int minRight;

            if (i == nums1Size) {

                minRight = nums2[j];

            } else if (j == nums2Size) {

                minRight = nums1[i];

            } else {

                minRight = (nums1[i] < nums2[j]) ? nums1[i] : nums2[j];

            }

            return (maxLeft + minRight) / 2.0;

        }

    }

    return 0.0;

}

int main() {

    int nums1[] = {1, 3};

    int nums2[] = {2};

    int nums1Size = sizeof(nums1) / sizeof(nums1[0]);

    int nums2Size = sizeof(nums2) / sizeof(nums2[0]);

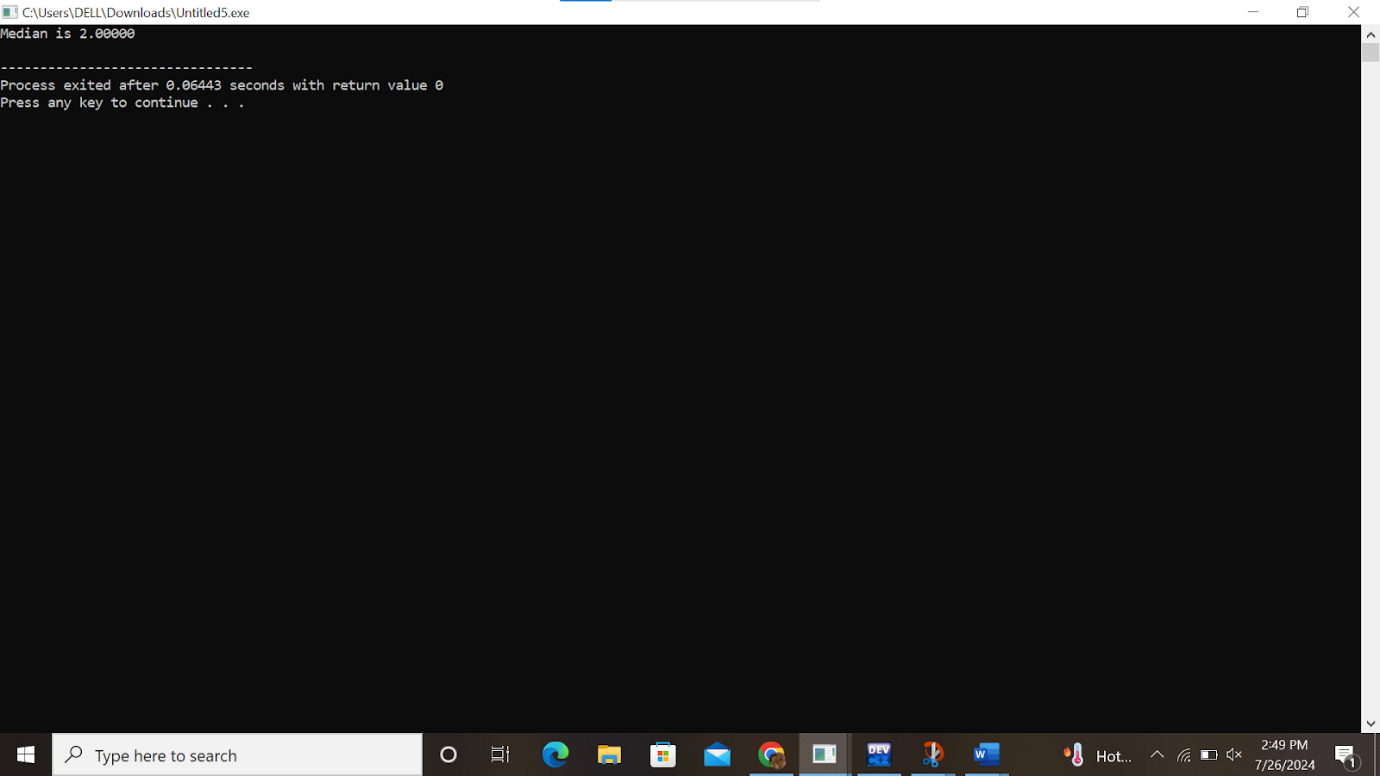
    double median = findMedianSortedArrays(nums1, nums1Size, nums2, nums2Size);

    printf("Median is %.5f\n", median);

    return 0;

}

**RESULT**

****

**COMPLEXITY ANALYSIS**

When analyzing the complexity of the algorithm for finding the median of two sorted arrays, we focus on both time and space complexities:

**Time Complexity**

1. **Binary Search on the Smaller Array:**
   * The algorithm uses binary search on the smaller of the two arrays (nums1 and nums2). Let’s denote the lengths of nums1 and nums2 as m and n respectively.
   * The binary search operates in O(log(min(m, n))) time, because it only searches within the smaller array. This is the core operation in determining the median.
2. **Overall Time Complexity:**
   * Since binary search is performed on the smaller array, the time complexity is O(log(min(m, n))). However, given that min(m, n) is always less than or equal to m + n, this can be simplified to O(log(m+n)). This notation provides an upper bound on the time complexity and aligns with the problem's requirement.

**Space Complexity**

1. **Auxiliary Space:**
   * The algorithm does not require additional space proportional to the size of the input arrays. It uses a few extra variables for tracking partition indices and boundary values.
   * Therefore, the space complexity is O(1), meaning it operates in constant space beyond the input arrays.

**Detailed Breakdown**

1. **Binary Search:**
   * Binary search in one array with length m takes O(log(m)) time.
   * Given the problem involves two arrays, the time complexity is O(log(min(m, n))) where min(m, n) represents the length of the smaller array. This is efficient compared to a straightforward merge-and-find approach which would take O(m+n) time.
2. **Handling Edge Cases:**
   * Special cases such as arrays with different sizes, empty arrays, or arrays where all elements are smaller or larger compared to the other array are handled by the algorithm efficiently within the same O(log(min(m, n))) complexity.

Summary

* Time Complexity: O(log(min(m, n))) ≈ O(log(m + n))
* Space Complexity: O(1)

The algorithm efficiently computes the median by leveraging binary search on the smaller array, making it well-suited for large datasets where merging the arrays would be impractical.

To understand the complexity of finding the median of two sorted arrays using the binary search approach, let's analyze the best case, worst case, and average case scenarios:

**BEST CASE**

Scenario: The best case occurs when the median can be found immediately in the first partition attempt. This happens when the binary search partitioning perfectly divides the combined arrays into two halves where all elements on the left side are less than or equal to all elements on the right side.

**Complexity:**

* Time Complexity: O(1). If the partitions are correct on the first attempt, no further iterations of binary search are required.

**Example:** Given nums1 = [1] and nums2 = [2], the partitions will immediately give:

* Left half: [1] (from nums1)
* Right half: [2] (from nums2) The median is directly found as the average of the maximum of the left half and the minimum of the right half.

**WORST CASE**

Scenario: The worst case occurs when the binary search needs to explore the entire range of the smaller array before finding the correct partition. This happens when the partitions are not correct until the final steps of binary search.

**Complexity:**

* Time Complexity: O(log(min(m, n))). In practice, this is equivalent to O(log(m+n)) since the binary search is performed on the smaller array, but its complexity still depends on the length of that smaller array.

Example: Consider nums1 = [1, 2, 3, 4, 5] and nums2 = [6, 7, 8, 9, 10]. The binary search will need to adjust multiple times before finding the correct partitions that satisfy the median condition.

**AVERAGE CASE**

Scenario: The average case considers the typical performance of the algorithm over a range of inputs. On average, the binary search will require a logarithmic number of steps proportional to the length of the smaller array.

**Complexity:**

* Time Complexity: O(log(min(m, n))). This average case is derived from the typical behavior of binary search, where the search space is halved in each step until the correct partition is found.

Example: Given nums1 = [1, 3, 5, 7, 9] and nums2 = [2, 4, 6, 8, 10], the binary search will usually converge to the correct partition after a logarithmic number of steps relative to the length of the smaller array.

**Best Case:** O(1) – When the correct partitions are found immediately.

**Worst Case:** O(log(min(m, n))) – When binary search needs to explore the entire range of the smaller array.

**Average Case:** O(log(min(m, n))) – Typical performance for binary search over random inputs.

**CONCLUSION**

The problem of finding the median of two sorted arrays efficiently is addressed using a binary search-based algorithm. The key insights and benefits of the approach are as follows:

1. **Optimal Time Complexity:** The proposed solution achieves an overall time complexity of O(log(m+n)), where m and n are the lengths of the two sorted arrays, nums1 and nums2, respectively. This efficiency is critical for handling large datasets, as it avoids the O(m+n) complexity of merging and then finding the median.
2. **Binary Search Technique**: By performing a binary search on the smaller of the two arrays, the algorithm efficiently partitions both arrays into left and right halves. This partitioning ensures that the elements in the left half are less than or equal to the elements in the right half, which allows for the accurate calculation of the median.
3. **Handling Edge Cases:** The algorithm gracefully handles edge cases such as when one of the arrays is empty, or when all elements of one array are less than or greater than the elements of the other array. This is done by setting boundary values to negative or positive infinity as needed.
4. **Space Efficiency:** The algorithm operates in O(1) space complexity, meaning it uses a constant amount of extra space beyond the input arrays. This is advantageous compared to other methods that might require additional storage for merged arrays or intermediate results.
5. **Accuracy:** The solution provides an accurate median by directly using the values from the partitions without the need for full array merging. It correctly handles both odd and even total lengths of the combined arrays.
6. **Practical Implementation:** The algorithm's efficiency and accuracy make it suitable for practical applications, such as real-time systems where performance is crucial, or for large-scale data analysis where merging large datasets is impractical.

In summary, the binary search-based method for finding the median of two sorted arrays is both theoretically sound and practically efficient. It effectively balances the need for speed and minimal space usage, providing a robust solution for a fundamental problem in computer science and data processing.