CAPSTONE PROJECT

PROJECT TITLE : MEDIAN OF TWO SORTED ARRAYS

CSA0637 - Design and Analysis of Algorithms for Dynamic Programming

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING

Supervisor

M. Joy Priyanka

DONE BY

E. Pranith Reddy (192210102)

P. Vedha Vyas (192210504)

APRIL 2024

Problem statement

The problem statement for finding the median of two sorted arrays typically goes like this: "Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be O(log(m+n)). "In simpler terms, you're tasked with finding the median value, which is the middle value in a sorted array if the array length is odd, or the average of the two middle values if the length is even, when you merge two sorted arrays. The complexity constraint of O(log(m+n)) .

Proposed design work

1.Identifying key component

- **Initialize variables**: Start by initializing variables for the lengths of the two arrays, m and n. Determine which array is shorter (if any) and store their lengths. Also, initialize variables to represent the start and end indices of the partitions for both arrays.

- **Partitioning:** Partition both arrays into two halves at positions i and j. For example, in nums1, divide it into nums1[0:i-1] and nums1[i:m-1], and similarly for nums2 into nums2[0:j-1] and nums2[j:n-1]

- **Check condition:** Ensure that the elements on the left side of the partition in nums1 and nums2 are less than or equal to the elements on the right side. This condition ensures that the median is correctly positioned.

- **Calculate median:** If the total length of the merged arrays is odd, the median is simply the maximum of the elements on the left side. If it's even, then the median is the average of the maximum of the left elements and the minimum of the right elements.

- **Adjust partition:** If the condition from step 3 is not satisfied, adjust the partition accordingly. If the median falls in the first half, move the partition to the right. If it falls in the second half, move the partition to the left. Repeat this process until the condition is met.

- **Repeat binary search**: Repeat steps 2-5 until you find the correct partition that satisfies the condition.

- **Return median:** Once the correct partition is found, return the median value.

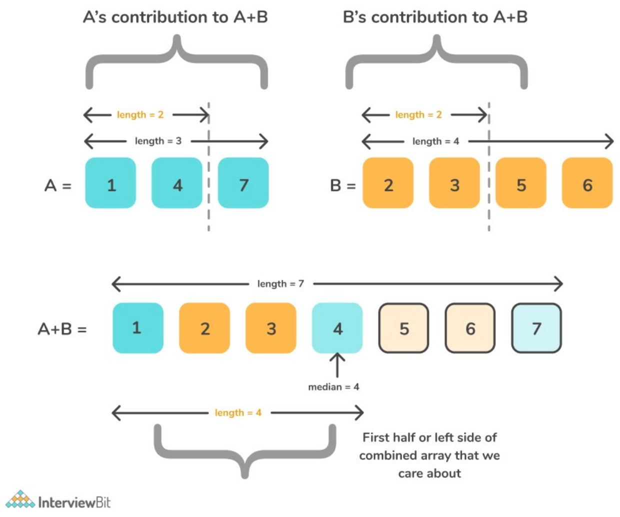
2. functionality

- **Merge the Arrays:** Combine the two sorted arrays into a single sorted array. This step ensures that all the elements are in sorted order.

- **Efficient Approach:** To achieve the desired time complexity of O(log(m+n)), where m and n are the lengths of the two arrays, you can use a binary search-based approach. This approach aims to find the partition points in the arrays such that elements on the left side of the partition are smaller or equal to elements on the right side.

- **Complexity:** The binary search-based approach ensures that the algorithm's time complexity remains within the specified bounds.

3.Architectural design



UI Design

1.Layout Design

- **Complexity Analysis:** Discuss the time complexity of the algorithm, which is O(log(min(m, n))). Explain how the binary search approach achieves the desired complexity.

- **Adjusting Binary Search Range:** If the condition is not met, adjust the binary search range based on the comparison of maximums and minimums.

- **Handling Edge Cases:** Deal with scenarios where one or both arrays are empty or the total length is odd.

- **Return Median:** Return the calculated median value.

2. feasible elements used

- **Binary Search Iteration:** The binary search algorithm iteratively adjusts the feasible element (partition point) until it satisfies the condition for the median. This process involves updating the feasible element based on comparisons with elements in both arrays.

- **Partitioning Arrays:** The feasible element divides each array into two parts: elements less than or equal to the feasible element and elements greater than the feasible element. These partitions are crucial for ensuring that the median condition is met.
- **Checking Median Condition**: The condition for the median involves verifying that elements on the left side of the partition points in both arrays are smaller or equal to elements on the right side. The feasible element helps determine whether this condition is satisfied.

- **Adjusting Binary Search Range:** If the condition for the median is not met, the binary search algorithm adjusts the feasible element to narrow down the search range. This adjustment ensures that the search converges to the correct partition points efficiently.

3.Element functions

- **Accessing Elements:** Given the sorted nature of the arrays, accessing elements by index can be done in constant time, O(1), providing direct access to any element.

- **Comparing Elements:** Comparing two elements can be done in constant time, O(1), by using basic comparison operators (<, >, ==).

- **Calculating Median:** The median calculation involves determining the middle element(s) of the merged arrays. This can be done in constant time, O(1), once the partition points are established.

- **Updating Partition Points:** Updating the partition points involves moving them closer to the target value efficiently. This typically takes logarithmic time, O(log(m)) or O(log(n)), depending on the size of the array being partitioned.

- **Checking Median Condition:** The median condition ensures that elements on the left side of the partitions are less than or equal to elements on the right side. This verification can be done in constant time, O(1), once the partition points are determined.

- **Adjusting Search Range:** Adjusting the search range involves narrowing down the possible locations for the partition points. This operation typically takes logarithmic time, O(log(m)) or O(log(n)), depending on the size of the search space.

- **Input:** Two sorted arrays, nums1 and nums2, of lengths m and n respectively.

- **Output:** The median value of the combined array formed by merging nums1 and nums2.

- **Result:** If the total number of elements (m + n) is odd, the median is the middle element of the combined array.

If the total number of elements is even, the median is the average of the two middle elements of the combined array.

Code

```
#include <stdio.h>


double findMedianSortedArrays(int* nums1, int nums1Size, int* nums2, int
nums2Size) {
   if (nums1Size > nums2Size) {
      return findMedianSortedArrays(nums2, nums2Size, nums1, nums1Size);
   }


   int m = nums1Size;
   int n = nums2Size;
   int total_length = m + n;
```

```
    int is_even = total_length % 2 == 0;
    int left_half_length = total_length / 2;

    int start = 0;
    int end = m;

    while (start <= end) {
        int partition_nums1 = (start + end) / 2;
        int partition_nums2 = left_half_length - partition_nums1;

        int nums1_left_max = (partition_nums1 == 0) ? INT_MIN :
nums1[partition_nums1 - 1];
        int nums1_right_min = (partition_nums1 == m) ? INT_MAX :
nums1[partition_nums1];

        int nums2_left_max = (partition_nums2 == 0) ? INT_MIN :
nums2[partition_nums2 - 1];
        int nums2_right_min = (partition_nums2 == n) ? INT_MAX :
nums2[partition_nums2];

        if (nums1_left_max <= nums2_right_min && nums2_left_max <=
nums1_right_min) {
            if (is_even) {
                return (double)(fmax(nums1_left_max, nums2_left_max) +
fmin(nums1_right_min, nums2_right_min)) / 2.0;
            } else {
                return fmin(nums1_right_min, nums2_right_min) > fmax(nums1_left_max,
nums2_left_max) ? fmin(nums1_right_min, nums2_right_min) :
fmax(nums1_left_max, nums2_left_max);
            }
        } else if (nums1_left_max > nums2_right_min) {
            end = partition_nums1 - 1;
        } else {
            start = partition_nums1 + 1;
        }
    }
    return -1;  // Error case
}
int main() {
    int nums1[] = {1, 3};
    int nums1Size = sizeof(nums1) / sizeof(nums1[0]);
    int nums2[] = {2};
    int nums2Size = sizeof(nums2) / sizeof(nums2[0]);

    double median = findMedianSortedArrays(nums1, nums1Size, nums2, nums2Size);
    printf("Median: %.5f\n", median);

    return 0;
}
```

Input: nums1 = [1,3], nums2 = [2]

Output: 2.00000

Explanation: Merged array = [1,2,3] and median is 2.

Conclusion

In conclusion, finding the median of two sorted arrays is a fundamental problem in computer science and mathematics. Through the use of efficient algorithms like the one demonstrated here, we can achieve this task with a time complexity of O(log(m + n)), where m and n are the lengths of the two input arrays. The algorithm utilizes binary search to iteratively adjust partition points in one of the arrays, ensuring that elements on the left side are less than or equal to elements on the right side. By carefully managing these partition points, we can efficiently locate the median value of the combined array formed by merging the two input arrays.