CAPSTONEPROJECT

# MEDIAN OF TWO SORTED ARRAYS USING DIVIDE AND CONQUER METHOD

**CSA0695-** DESIGN ANALYSIS AND ALGORITHMS FOR OPEN ADDRESSING TECHNIQUES

SAVEETHA SCHOOL OF ENGINEERING

SIMATS ENGINEERING



Supervisor

Dr. R. Dhanalakshmi

Done by

B.THARUN KUMAR 192210232

# MEDIAN OF TWO SORTED ARRAYS

**PROBLEM STATEMENT:**

Median of Two Sorted Arrays 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)). Example 1: Input: nums1 = [1,3], nums2 = [2] Output: 2.00000 Explanation: merged array = [1,2,3] and median is 2.

**ABSTRACT:**

To solve the problem of checking whether a given string s can be split into two or more non empty substrings with numerical values in descending order and each pair of adjacent values differing by exactly 1, can follow this approach: 1.Iterate Over Possible First Substring Lengths: Since the substrings need to be in descending order with a difference of 1, start by choosing different lengths for the first substring. Convert this substring into an integer. 2.Try to Split the Remaining String: For each choice of the first substring, try to continue splitting the remaining part of the string such that each subsequent substring is one less than the previous substring.

**INTRODUCTION:**

The problem of finding the median of two sorted arrays using the divide and conquer approach is an extension of binary search and requires efficient algorithms to solve it in logarithmic time. The task is to find the median of two sorted arrays, typically denoted as nums1 and nums2, both of sizes m and n, respectively. The brute-force approach would involve merging the two arrays into one and then finding the median, but that would take O(m + n) time. Instead, the divide and conquer or binary search approach allows us to find the median in O(log(min(m, n))) time.

Time Complexity: Since we perform binary search on the smaller array, the time complexity is O(log(min(m, n))), which is very efficient compared to a brute-force approach. This approach is optimal for large datasets and is widely regarded for its logarithmic efficiency.

**CODING:**

To solve the problem, we use a that simulates the concept of minimizing product selection for an e-commerce platform. This example assumes that you have product categories, and based on user input (category and price range), it filters and narrows down product selections. This is a simple illustration, which can be expanded with more complex filtering and recommendation logic as needed.

**C-programming**

#include <iostream>

#include <vector>

#include <climits>

using namespace std;

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) { if (nums1.size() > nums2.size()) {

return findMedianSortedArrays(nums2, nums1);

}

int x = nums1.size(); int y = nums2.size(); int low = 0, high = x; while (low <= high) {

int partitionX = (low + high) / 2;

int partitionY = (x + y + 1) / 2 - partitionX;

int maxX = (partitionX == 0) ? INT\_MIN : nums1[partitionX - 1]; int maxY = (partitionY == 0) ? INT\_MIN : nums2[partitionY - 1]; int minX = (partitionX == x) ? INT\_MAX : nums1[partitionX]; int minY = (partitionY == y) ? INT\_MAX : nums2[partitionY];

if (maxX <= minY && maxY <= minX) { if ((x + y) % 2 == 0) {

return (double)(max(maxX, maxY) + min(minX, minY)) / 2;

} else {

return (double)max(maxX, maxY);

}

}

int main() {

vector<int> nums1 = {1, 3}; vector<int> nums2 = {2};

double median = findMedianSortedArrays(nums1, nums2); cout << "The median is: " << median << endl;

return 0;

}

**OUTPUT:**



**COMPLEXITY ANALYSIS:**

To evaluate the complexity of the problem of splitting a string sss into descending consecutive values, we analyze the best case, worst case, and average case scenarios.

Best Case

Scenario: The best case occurs when the string can be quickly identified as either valid or invalid with minimal checks. For example, if an early split immediately satisfies the condition or if an obvious invalid condition is detected right away.

Time Complexity: O(n)O(n)O(n)

* In this scenario, the function might need to make only a single pass through the string or evaluate a very few number of splits to conclude the result.

Explanation: If a valid split is found quickly or the string is evidently invalid from an early check, the operations performed are linear, resulting in O(n)O(n)O(n) complexity.

.

Worst Case

Scenario: The worst case happens when we need to explore all possible ways to split the string into substrings to determine that no valid split exists. This involves evaluating every potential split configuration.

Time Complexity: O(2n)O(2^n)O(2n)

* This is because each digit of the string can either continue the current substring or start a new one, leading to an exponential number of split possibilities.

Explanation: Given a string of length nnn, the number of ways to split it into substrings grows exponentially as each position in the string offers a binary decision, leading to

O(2n)O(2^n)O(2n) possible splits.

Average Case

Scenario: The average case complexity considers more typical scenarios where the string is split into substrings in a balanced manner, without needing exhaustive checking as in the worst case.

Time Complexity: O(n⋅2n/2)O(n \cdot 2^{n/2})O(n⋅2n/2)

* On average, we may expect to evaluate splits in a balanced way, checking a reasonable number of configurations that are more than linear but not as exhaustive as the worst case.

Explanation: The average case assumes that the number of split points checked is more balanced, considering a midway between the quick checks of the best case and the exhaustive checks of the worst case.

This complexity analysis highlights the computational challenges in splitting a string into valid descending consecutive values, emphasizing the importance of efficient algorithm design to handle the exponential nature of the problem in the worst-case scenarios.

**FUTURESCOPE:**

The divide and conquer approach to ﬁnding the median of two sorted arrays has various applications, such as 1) Data analysis,2) Signal processing,3) Machine learning. Additionally, there are further optimizations and variations of this algorithm that can be explored to enhance its performance in speciﬁc scenarios.

## CONCLUSION

## In conclusion, the problem of splitting a string into descending consecutive values poses computational challenges that vary based on the string's length nnn and the number of possible split configurations. The best-case scenario offers linear complexity O(n)O(n)O(n), where a valid split can be quickly identified with minimal checks. However, the worst-case scenario exhibits exponential complexity O(2n)O(2^n)O(2n), necessitating evaluation of every potential split to determine if no valid configuration exists. On average, the complexity balances between these extremes at O(n⋅2n/2)O(n \cdot 2^{n/2})O(n⋅2n/2), reflecting a more moderate

exploration of split possibilities. Efficiently solving this problem requires strategic algorithm design to manage the exponential growth in potential configurations, ensuring robust validation of descending order and consecutive difference criteria within substrings.