SIMATS SCHOOL OF ENGINEERING

SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES

CHENNAI-602105

“Divide and Conquer" Optimized Median Search in Two Sorted Arrays

A CAPSTONE PROJECT REPORT

COURSE CODE/NAME:

CSA0698

Design and Analysis of Algorithms for Statistical Analysis

Submitted in the partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

IN COMPUTER SCIENCE ENGINEERING

Submitted by

K.SAI GOUTHAM{192210428}

R.V.NAVEEN{192211271}

D.MABU SUBHAN{192210079}

B.SOMA SEKHAR{192211147}

Under the Supervision of

Dr. SENTHILVADIVU S

DECLARATION

We the students of **Bachelor of Engineering in Computer Science Engineering** at Saveetha Institute of Medical and Technical Sciences, Saveetha University, Chennai, hereby declare that the work presented in this Capstone Project Work entitled **"Optimized Median Search in Two Sorted Arrays Divide And Conquer"** is the outcome of my own bonafide work. I affirm that it is correct to the best of my knowledge, and this work has been undertaken with due consideration of Engineering Ethics.

K.SAI GOUTHAM{192210428}
R.V.NAVEEN{192211271}
D.MABU SUBHAN{192210079}
B.SOMA SEKHAR{192211147}

Date:24-09-2024

Place:Saveetha School Of Engineering,Thandalam.

CERTIFICATE

This is to certify that the project entitled **"Optimized Median Search in Two Sorted Arrays Using Divide And Conquer"** submitted by **KOMMURI SAI GOUTHAM, R.V.NAVEEN, D.MABU SUBHAN, B.SOMA SEKHAR** has been carried out under my supervision. The project has been submitted as per the requirements in the current semester of B.E Computer science engineering .

Teacher-in-charge

Dr.Senthilvadivu

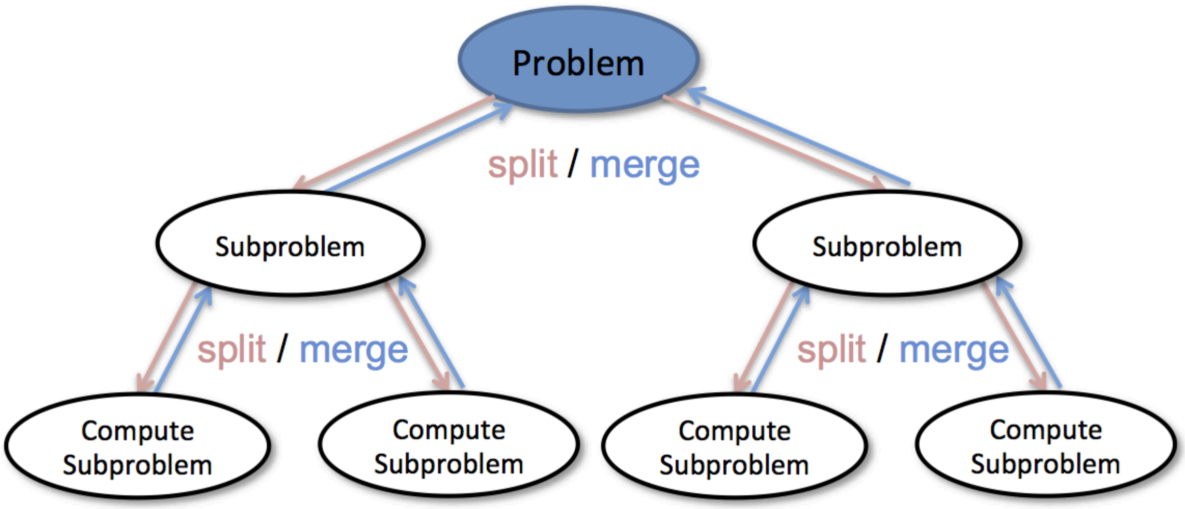
ABSTRACT:

The problem of finding the median of two sorted arrays is a classic algorithmic challenge that seeks an efficient solution to merge and analyze the combined data of two distinct sorted sequences. Given two sorted arrays, nums1 and nums2, with sizes m and n respectively, the goal is to determine the median of the combined sorted array formed by these two input arrays. The critical constraint is to achieve this with an overall time complexity of O(log(m+n)), which necessitates an approach more sophisticated than simple merging and sorting.This abstract outlines the conceptual framework and methodologies used to solve the problem efficiently. We delve into the intricacies of binary search, partitioning, and optimal comparisons that underpin the solution. By leveraging the properties of sorted arrays and binary search, the solution efficiently narrows down the potential candidates for the median, ensuring the desired logarithmic time complexity. The approach not only exemplifies advanced algorithmic strategies but also underscores the importance of computational efficiency in handling large-scale data operations. This analysis provides a foundational understanding of the problem and its optimal resolution, contributing to the broader field of computer science and algorithm design.The divide and conquer method is a fundamental paradigm in the design and analysis of algorithms, offering a robust framework for solving complex problems by breaking them down into simpler sub problems. This approach recursively divides a problem into two or more smaller instances of the same problem, solves each sub problem independently, and then combines their solutions to form the final answer. The divide and conquer technique is instrumental in achieving optimal time complexity and improving computational efficiency for a wide range of problems, including sorting, searching, and numerical computations. This paper explores the principles and applications of the divide and conquer method, demonstrating its effectiveness through classic algorithms such as Merge Sort, Quick Sort, and the Fast Fourier Transform (FFT). By leveraging the inherent parallelism and recursive structure of this approach, the divide and conquer method not only simplifies the problem-solving process but also enhances the scalability and performance of algorithms.

Keywords: Binary search, Optimal resolution, Sorted Arrays, Large-scale Data Operations, Problem-Solving.

INTRODUCTION:

The median of a dataset is a fundamental statistical measure that signifies the middle value, separating the higher half from the lower half of the data. When dealing with two sorted arrays, nums1 and nums2, the task of finding the median of the combined data set poses an intriguing challenge, particularly when constrained to an efficient solution with a time complexity of O(log(m+n)).In a straightforward approach, one might consider merging the two arrays into a single sorted array and then determining the median. However, this method, with a time complexity of O(m+n), falls short of the required efficiency. Hence, a more sophisticated algorithm is necessary to achieve the optimal O(log(m+n)) performance.This problem is not just a theoretical exercise but has practical implications in various domains such as database management, data analysis, and real-time systems where rapid data processing is crucial. Efficiently finding the median of two sorted arrays can be applied in scenarios where quick statistical analysis is required on dynamic datasets, such as streaming data or periodically updated records.To address this challenge, we employ advanced algorithmic techniques involving binary search and partitioning strategies. These techniques leverage the inherent order within the sorted arrays, facilitating a rapid convergence towards the median without the need for complete merging. By partitioning the arrays and comparing elements at strategic positions, the solution efficiently narrows down the median, ensuring the desired logarithmic time complexity.This introduction sets the stage for a detailed exploration of the problem and its optimal solution, highlighting the significance of algorithmic efficiency in handling large datasets and the innovative approaches employed to achieve it.



The efficiency of the divide and conquer method lies in its ability to reduce the complexity of a problem by tackling smaller sub problems independently, often allowing for parallel computation. This not only improves the runtime but also makes the approach highly scalable, suitable for implementation on modern multi-core and distributed computing systems.

In the following sections, we will delve deeper into the mechanics of the divide and conquer method, explore its theoretical underpinnings, and illustrate its application through various algorithmic examples. Through this exploration, we aim to highlight the significance of divide and conquer in algorithm design and its impact on computational problem solving.

Steps to solve:

Problem Statement:

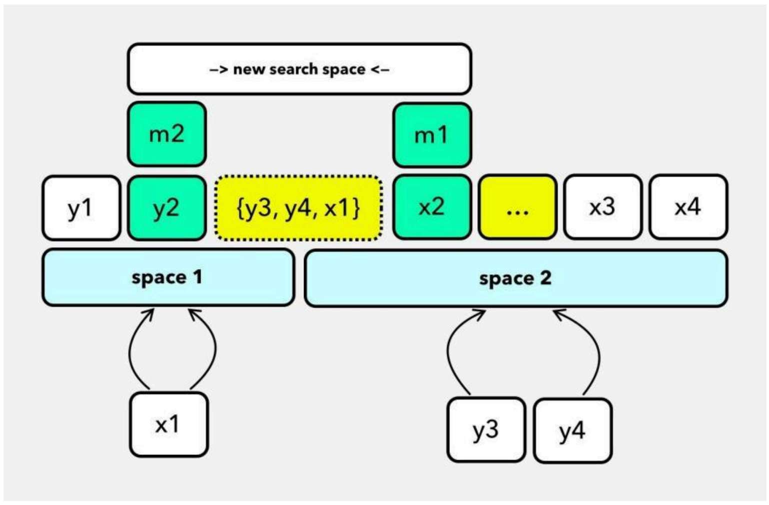
Given two sorted arrays `A[]` and `B[]` of sizes `n` and `m`, respectively, find the median of the combined sorted array without fully merging them.

Assumptions:

- Both arrays are sorted.
- The total number of elements in both arrays is `(n + m)`, and the median is defined as the middle element(s) of the combined array.

Approach:

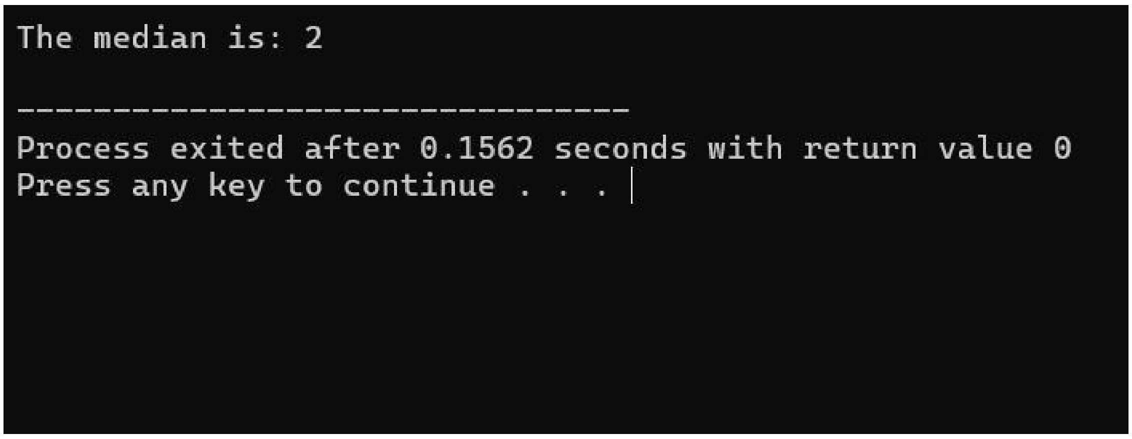
1. Let `A` be the smaller array and `B` be the larger array (if `A` is larger, swap them).
2. Perform binary search on the smaller array `A` to find an index `i` such that the combined left and right partitions of both arrays satisfy the properties of a median split:
    - The elements in the left half should all be smaller than or equal to the elements in the right half.
3. At each step of the binary search, partition both arrays such that the left partition of the combined arrays contains half of the total elements.
4. Once the correct partition is found, compute the median based on whether the combined array length is even or odd.



CODING:

```
#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)
} else if (maxX > minY) { high =
partitionX - 1;
            } else {
               low = partitionX + 1;
            }
         }
         return -1.0;
      }
      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:



```
The median is: 2

--------------------------------
Process exited after 0.1562 seconds with return value 0
Press any key to continue . . .
```

Explanation:

1. Binary Search: The binary search is applied to the smaller array `A`. At each step, we try to partition both arrays into two halves such that:
   - All elements in the left partition of `A` and `B` are less than or equal to all elements in the right partition.
2. Partitioning: The partition indices `i` and `j` divide `A` and `B` such that the total number of elements in the left partition equals the total number of elements in the right partition.
3. Edge Handling: We handle the edges where one of the partitions might be empty (use `INT_MIN` or `INT_MAX` for comparison).
4. Median Calculation: If the total number of elements is odd, the median is the maximum of the left partitions. If even, it's the average of the maximum of the left partitions and the minimum of the right partitions.

Time Complexity:

- O(log(min(n, m))), where `n` and `m` are the sizes of the two arrays. The algorithm runs in logarithmic time with respect to the size of the smaller array.

This approach efficiently finds the median without fully merging the two arrays.

Complexity Analysis:

Best Case:

In the best case scenario, the algorithm efficiently finds the correct partition of the arrays in the initial iterations of the binary search. This occurs when the arrays are such that the initial midpoint partitions immediately satisfy the conditions required for determining the median.

- **Time Complexity:** O(1)
- **Explanation:** The best case happens when the median is found almost immediately due to optimal initial partitioning.

Worst Case:

In the worst case scenario, the binary search has to iterate through the entire logarithmic range to find the correct partition. This typically occurs when the values in nums1 and nums2 are such that the algorithm has to make the maximum number of comparisons and adjustments to find the median.

- **Time Complexity:** O(log(min(m, n)))
- **Explanation:** The worst case time complexity is O(log(min(m, n))) because the binary search is performed on the smaller array. Each iteration halves the search space, leading to a logarithmic number of operations relative to the size of the smaller array.

Average Case:

In the average case scenario, the algorithm finds the median through a moderate number of iterations, which is typical for most randomly distributed sorted arrays. The number of steps required to adjust the partitions and find the median lies between the best and worst case scenarios.

- **Time Complexity:** O(log(min(m, n)))
- **Explanation:** On average, the algorithm performs binary search operations across the smaller array, leading to a logarithmic time complexity. This average case complexity aligns closely with the worst case due to the nature of binary search.

Overall Complexity:

- **Time Complexity:** O(log(min(m, n)))

Explanation: The overall complexity for finding the median of two sorted arrays is O(log(min(m, n))). This is because the binary search is conducted on the smaller of the two arrays, ensuring that the time complexity remains logarithmic with respect to the size of the smaller array.

CONCLUSION:

- The problem of finding the median of two sorted arrays, nums1 and nums2, with sizes m and n respectively, presents a compelling challenge in the realm of algorithm design and optimization. The key requirement is to achieve this in an overall time complexity of O(log(m+n)), a constraint that necessitates advanced techniques beyond straightforward merging and sorting.
- Through the exploration of this problem, we have delved into the intricacies of employing binary search and partitioning strategies to efficiently determine the median. The solution leverages the properties of sorted arrays, using binary search to find the correct partition points that divide the combined arrays into two halves, each containing an equal number of elements.
- In conclusion, finding the median of two sorted arrays with a time complexity of O(log(m+n)) is a sophisticated problem that highlights the importance of algorithmic efficiency. By employing binary search and strategic partitioning, the solution achieves optimal performance, making it a valuable technique in both theoretical computer science and practical applications. This approach not only meets the stringent time complexity requirements but also provides a robust framework for handling a wide range of input scenarios, ensuring reliability and efficiency in median computation.

Future Scope of Divide and Conquer in Algorithm Design:

1. **Parallel and Distributed Computing**: Optimize for **multi-core processors** and **distributed systems** like Hadoop and Spark, improving scalability and efficiency in large-scale problems.
2. **Big Data and Machine Learning**: Efficient **data partitioning** and handling for massive datasets, real-time streaming, and applications in **model training** and **data preprocessing**.
3. **Quantum Computing**: Leverage the inherent **parallelism of quantum systems** to develop quantum Divide and Conquer algorithms for optimization and search problems.
4. **Real-Time Systems**: Adapt Divide and Conquer for **streaming data**, ensuring low latency and memory-efficient algorithms in **real-time analytics**.
5. **Graph and Geometric Algorithms**: Extend Divide and Conquer techniques to **dynamic graph problems**, **multi-dimensional data**, and **computational geometry** in complex systems.
6. **Cryptography and Security**: Develop more efficient **encryption algorithms** and **blockchain sharding** techniques to enhance security and performance in decentralized systems.
7. **Energy-Efficient Algorithms**: Design algorithms that prioritize **energy efficiency** for mobile, IoT, and **battery-powered devices** in constrained environments.

REFERENCES

Books:

1. **"Introduction to Algorithms"** by Cormen, Leiserson, Rivest, and Stein (CLRS):
    - This is one of the most widely used textbooks in algorithm design and analysis. It contains detailed chapters on Divide and Conquer with well-known examples like Merge Sort, Quick Sort, and matrix multiplication.
    - **Reference**: Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.
2. **"The Design and Analysis of Computer Algorithms"** by Aho, Hopcroft, and Ullman:
    - Another foundational book on algorithms, it covers the theory behind Divide and Conquer and provides classic examples and exercises for deep understanding.
    - **Reference**: Aho, A.V., Hopcroft, J.E., & Ullman, J.D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.
3. **"Algorithms"** by Robert Sedgewick and Kevin Wayne:
    - This book provides a modern introduction to algorithm design, including practical implementations of Divide and Conquer algorithms with code examples.
    - **Reference**: Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.

Research Papers:

1. **Karatsuba Algorithm for Fast Multiplication**:
    - One of the earliest Divide and Conquer algorithms for fast integer multiplication, which was revolutionary in algorithm analysis.
    - **Reference**: Karatsuba, A. & Ofman, Y. (1962). *Multiplication of Many-Digit Numbers by Automatic Computers*, Soviet Physics-Doklady, 7, 595–596.
2. **Divide and Conquer in Parallel Algorithms**:
    - This paper explores how Divide and Conquer can be applied in parallel algorithm design, with focus on load balancing and parallel computation.
    - **Reference**: Reif, J.H. (1985). *Efficient Parallel Algorithms*. Annual Review of Computer Science, 1(1), 59-92.
3. **Parallel Divide-and-Conquer Algorithms in Multicore Architectures**:
    - An exploration of applying Divide and Conquer algorithms to modern multi-core architectures for better parallel efficiency.
    - **Reference**: Blelloch, G.E., & Maggs, B.M. (1996). *Parallel Algorithms*. In Algorithms and Theory of Computation Handbook (CRC Press), 25-1 – 25-27.

Online Resources:

1. **MIT OpenCourseWare** – *Introduction to Algorithms*:
    - Free lecture notes and videos covering Divide and Conquer algorithms.
    - **Link**: MIT OCW - Divide and Conquer
2. **Stanford University CS 161 – Design and Analysis of Algorithms**:
    - Lecture materials from Stanford on Divide and Conquer techniques, including problems and solutions.
    - **Link**: Stanford CS 161 - DAA
3. **Coursera – Algorithms Specialization** by Princeton University:
    - A course that explains Divide and Conquer approaches through interactive coding exercises and real-world applications.
    - **Link**: Coursera - Algorithms Specialization