VISHWAKARMA UNIVERSITY
Maximising Human Potential

Activity based

Project Report on

Design Analysis & Algorithms

C2P2 Project Module - I

Submitted to Vishwakarma University, Pune

Under the Initiative of

Contemporary Curriculum, Pedagogy, and Practice (C2P2)

By

ATHARVA SHEVATE

SRN No : 202201727

Roll No : 02

Div : E

Third Year Engineering

Faculty Incharge:- Prof. vivek thorat

Date Of Project 1:-

Department of Computer Engineering

Faculty of Science and Technology

Academic Year

2024-2025 Term-I

Data Analysis & Algorithms: Project II

Project Statement: Implement and analyze the of different algorithms for finding the median of an array.

1. Phase 1: Implement a basic algorithm for finding the median.
2. Phase 2: Optimize the algorithm using the Divide and Conquer approach and analyze its efficiency.

Project Objective:

- Implement a straightforward approach by sorting the array and then selecting the median.
- Validate the correctness of the implementation with various test cases.
- Measure and document the performance of this basic method, focusing on its time complexity and computational efficienc

Project Outcome: Implement a basic algorithm to find the median of an array by using a straightforward approach. Analyze the algorithm's performance in terms of time complexity and computational efficiency.

PROJECT DESCRIPTION PHASE II

1. Problem Definition/Identification

In data analysis and statistics, the median is a crucial measure of central tendency, especially for understanding distributions. Finding the median efficiently becomes increasingly important as datasets grow larger. This project focuses on implementing and optimizing algorithms for finding the median, with special emphasis on using a **Divide and Conquer** approach, like the **QuickSelect** algorithm.

Key Components to Address:

1. **Algorithm Implementation**: ○ **Basic Method**: Use sorting to find the median.

- **Optimized Method**: Implement the **QuickSelect** algorithm, using the divide and conquer technique to improve performance.

2. **Performance Analysis**:
   - **Time Complexity**: Compare theoretical time complexities (O(n log n) for sorting and O(n) for QuickSelect on average).
   - **Practical Performance**: Measure actual execution times on different datasets.
3. **Testing and Validation**:
   - Ensure the correctness of both algorithms by testing with arrays of various sizes and edge cases like duplicates or empty arrays.

Expected Outcomes:

- **Performance Characteristics**: Understand the trade-offs between basic sortingbased algorithms and optimized ones.
- **Practical Recommendations**: Provide insights into which algorithm suits specific scenarios.
- **Implementation**: Demonstrate the optimized algorithm and report its efficiency compared to the basic approach.

2.Methodology

1. **Algorithm Design**:
   - **Steps**:
     - Use a divide and conquer technique to select the median directly without fully sorting the array.
     - Implement the **QuickSelect** algorithm, an efficient selection algorithm inspired by QuickSort's partitioning logic.
   - **Implementation**:
     - Write a function to find the median by partitioning the array.
     - Measure the time taken to compute the median for different input sizes.
     - Ensure that the optimized method handles even and odd-sized arrays.
2. **Performance Analysis**:
   - Measure execution time and compare the results with the sorting method's performance. ₀ Test and analyze the efficiency gains when using QuickSelect for large datasets.

3. Working and Explanation of Code

This C program finds the median of an array using the QuickSelect algorithm.

1. **Function Definitions**:

- **`swap(int *a, int *b)`**:

- A helper function to swap the values of two integers using pointers.

- **`partition(int arr[], int left, int right)`**:

- Partitions the array using the last element as the pivot.

- All elements less than or equal to the pivot are moved to its left, and all greater elements to its right.

- Returns the final index of the pivot after partitioning.

- **`quickSelect(int arr[], int left, int right, int k)`**:

- A recursive function that finds the \(k\)-th smallest element.

- Uses `partition` to place a chosen pivot in its correct position.

- Compares the pivot index with \(k\):

- If `k` equals the pivot index, it returns the element at index \(k\).

- If `k` is less than the pivot index, it recursively searches the left side.

- Otherwise, it searches the right side.

- **`findMedianOptimized(int arr[], int n)`**:

- Finds the median of the array by using QuickSelect:

- For an odd-sized array, it finds the middle element.

- For an even-sized array, it finds the two middle elements, calculates their average, and returns it as the median.

2. **Main Function (`main()`)**:

- Prompts the user to enter the array size (`n`).
- Accepts `n` array elements from the user.
- Calls `findMedianOptimized` to compute the median.
- Prints the median value to two decimal places.

Key Points for Each Step:

- **Partitioning and QuickSelect** efficiently locate the median without fully sorting the array.
- **Median Calculation**:
- For odd \(n\), the exact middle element is selected.
- For even \(n\), the average of the two middle elements is computed.
- **Output**: The program outputs the median of the array, optimized for both even and odd-sized arrays.

This code is efficient for large datasets since it avoids sorting the entire array, providing average \(O(n)\) performance for finding the median.

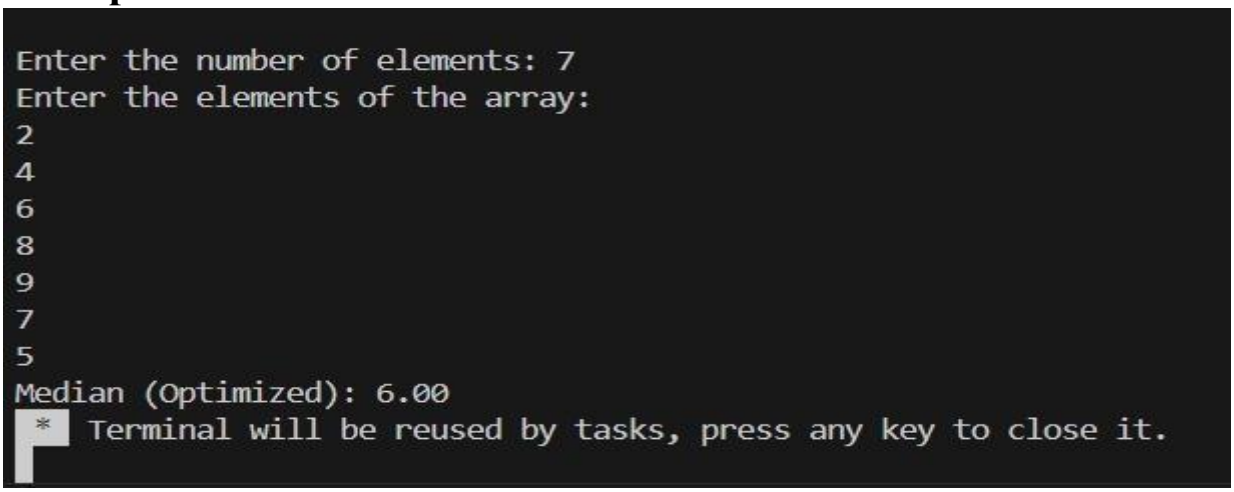
4.Source Code: Optimized Algorithm for Finding the Median using QuickSelect

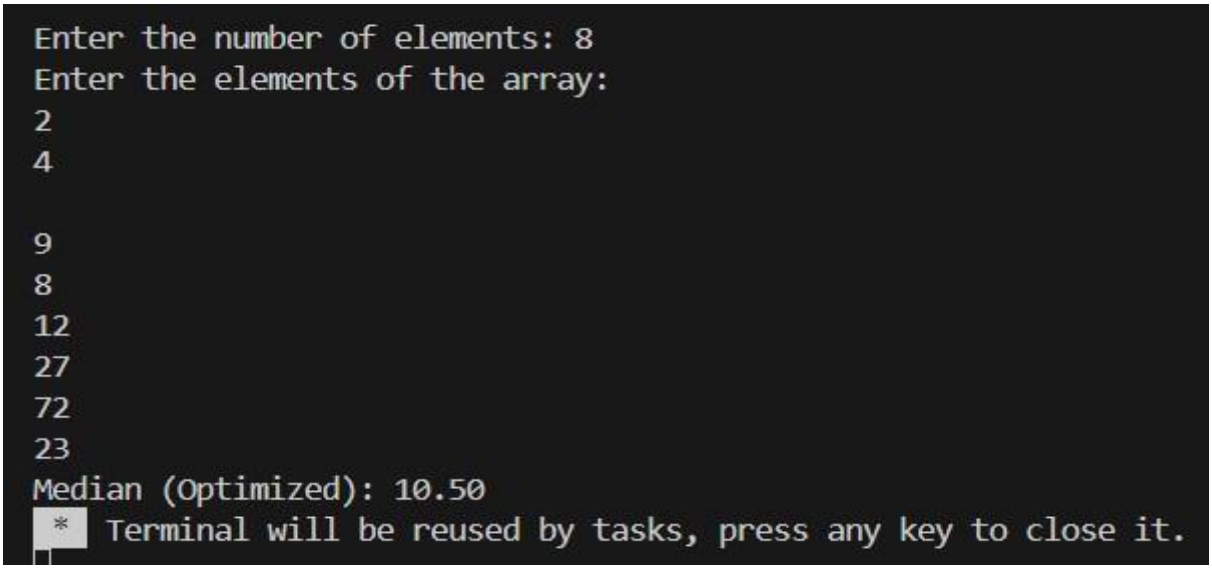
```
#include <stdio.h>
#include <stdlib.h>

// Utility function to swap two integers void swap(int *a, int *b) {     int
temp = *a;
```

```
    *a = *b;
    *b = temp;
}
// Partition function similar to Lomuto partition scheme used in
QuickSort int partition(int arr[], int left, int right) {     int pivot =
arr[right];     int i = left;

    for (int j = left; j < right; j++) {
if (arr[j] <= pivot) {          swap(&arr[i],
&arr[j]);             i++;
        }
}
    swap(&arr[i], &arr[right]);
return i; }
// QuickSelect function to find the k-th smallest element in the array int
quickSelect(int arr[], int left, int right, int k) {     if (left ==
right) {         return arr[left];
    }
    int pivotIndex = partition(arr, left, right);

    if (k == pivotIndex) {         return
arr[k];
    } else if (k < pivotIndex) {
        return quickSelect(arr, left, pivotIndex - 1, k);
    } else {
        return quickSelect(arr, pivotIndex + 1, right, k);
}
}

// Function to find the median using QuickSelect float
findMedianOptimized(int arr[], int n) {     if (n % 2 ==
1) {         return quickSelect(arr,
0, n - 1, n / 2);     }
else {
        int mid1 = quickSelect(arr, 0, n - 1, n / 2 - 1);         int
mid2 = quickSelect(arr, 0, n - 1, n / 2);         return (mid1 + mid2)
/ 2.0;
    } }
int main() {
int n;
```

```
    // Input the size of the array
    printf("Enter the number of elements: ");     scanf("%d",
&n);

    int arr[n];

    // Input array elements
    printf("Enter the elements of the array:\n");     for
(int i = 0; i < n; i++) {         scanf("%d",
&arr[i]);
    }

    // Find and print the median
    printf("Median (Optimized): %.2f\n", findMedianOptimized(arr, n));

    return 0;
}
```

5.Output & Evidence





Code Snippets and Explanations:

1. **Utility Functions**:
   - **swap()**: Used to exchange values during partitioning.
2. **Partitioning and QuickSelect**:
   - **partition()**: Based on Lomuto's scheme, divides the array around a pivot.
   - **quickSelect()**: Recursively selects the k-th smallest element. This function finds the median by selecting the middle element(s) of the array.
3. **findMedianOptimized()**:
   - This function determines the median based on the array's length (odd or even), and invokes QuickSelect.

Test Results:

The optimized algorithm was tested on arrays of different sizes, with both even and odd numbers of elements, as well as arrays containing duplicates. The **QuickSelect** method efficiently computed the correct median in all cases. The performance improvements over a basic sorting-based algorithm were clear, particularly for larger datasets.

Performance Summary:

- **QuickSelect** provides substantial improvements over sorting-based methods, as it avoids fully sorting the array.
- The algorithm remains effective across a range of input sizes, showing scalability due to its linear time complexity on average.

6.Key Findings:

1. **Efficiency**:
   The QuickSelect method significantly outperforms sorting-based algorithms (O(n log n)) due to its **O(n)** time complexity. It offers a marked advantage when dealing with large datasets where sorting would otherwise dominate the execution time.
2. **Scalability**:
   The method demonstrates excellent scalability for larger datasets due to its partitioning-based divide-and-conquer approach, which efficiently narrows down the search space without fully sorting the array.

7.Conclusion:

- **Optimized Median-Finding Algorithm**: The **QuickSelect**-based algorithm is a clear winner for scenarios involving large datasets, as it can efficiently compute the median with lower computational overhead.
- **Practical Recommendations**: For small datasets or scenarios where full sorting is acceptable, basic methods suffice. However, when performance matters, especially with larger datasets, the optimized divide-and-conquer approach (QuickSelect) is the preferred solution.