Name: Mrigaj Shaw
Roll No: 2024CSB041

Assignment 4

Problem 1

Problem Statement: An online educational analytics platform conducts national-level mock examinations for various competitive tests. In each examination session, approximately 110,000 students participate simultaneously. Once the test concludes, the scores obtained by all students are collected and stored on the server in the form of an unsorted integer array. To generate real-time performance analytics, such as identifying the central tendency of student performance, the platform needs to compute the median score as efficiently as possible. Since the dataset is large, the choice of algorithm has a significant impact on both execution time and memory utilization. Write a C program to determine the median score using two different approaches, as described below:

i) Fixed Pivot-Based Approach: Employs a divide-and-conquer approach in which the pivot is selected in a fixed manner.

ii) Quickselect-Based Approach: Determine the median using the Quickselect algorithm, which finds the required order statistic without fully sorting the array.

After implementing both methods, perform a comparative analysis of the two approaches by evaluating and discussing their:

i) Time complexity in the best case, average case, and worst case

ii) Space complexity in the best case, average case, and worst case

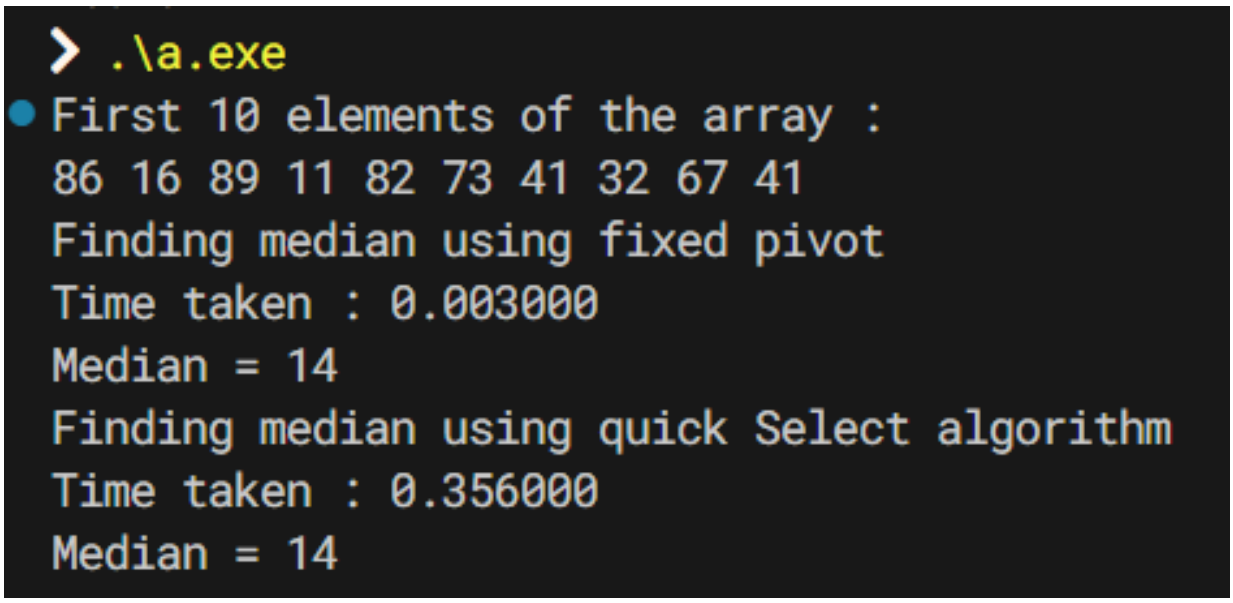
Finally, justify which approach is more suitable for handling large-scale, real-time examination data and explain your conclusion with appropriate reasoning.

Code

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

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int *a, int low, int high) {
    int pivot_index = high;
    int i = low - 1;
    int pivot = a[pivot_index];
    swap(&a[pivot_index], &a[high]);

    for (int j = low; j < high; j++) {
        if (a[j] <= pivot) {
            swap(&a[++i], &a[j]);
        }
    }
    swap(&a[++i], &a[high]);
    return i;
}

int fixedPivotMedian(int *a, int low, int high, const int medianIndex) {
    if(low <= high) {
        int partitionIndex = partition(a, low, high);
        if(partitionIndex == medianIndex) {
            return a[partitionIndex];
        }
        if(partitionIndex > medianIndex) return fixedPivotMedian(a, low,
   partitionIndex - 1, medianIndex);
```

```
        else if(partitionIndex < medianIndex) return fixedPivotMedian(a, partitionIndex + 1, high, medianIndex);
    }
    return -1;
}

int quickSelect(int *a, int low, int high , int kthSmallestElement) {
    if(kthSmallestElement > 0 && kthSmallestElement <= high - low + 1) {
        int index = partition(a, low, high);
        if(index - low == kthSmallestElement - 1) return a[index];

        if(index - low > kthSmallestElement - 1) return quickSelect(a, low, index - 1, kthSmallestElement);
        else return quickSelect(a, index + 1, high, kthSmallestElement - (index - low + 1));
    }
    return -1;
}

int main() {
    int n = (int) 1e5 + 1e4;
    int medianIndex = -1;
    clock_t start, end;
    int *a = (int *) malloc(n * sizeof(int));
    srand(time(NULL));
    for(int i = 0; i < n; i++) a[i] = rand() % 100;

    printf("First 10 elements of the array : \n");
    for(int i = 0; i < 10; i++) printf("%d ", a[i]);
    printf("\n");

    printf("Finding median using fixed pivot\n");
    start = clock();
    medianIndex = fixedPivotMedian(a, 0, n - 1, n / 2 + 1);
    end = clock();
    printf("Time taken : %f\n", (double)(end - start) / CLOCKS_PER_SEC);
    printf("Median = %d\n", a[medianIndex]);

    printf("Finding median using quick Select algorithm\n");

    start = clock();
    medianIndex = quickSelect(a, 0, n - 1, n / 2 + 1);
    end = clock();

    printf("Time taken : %f\n", (double)(end - start) / CLOCKS_PER_SEC);

    printf("Median = %d\n", a[medianIndex]);
    return 0;

}
```

Output



Inference

The following conclusions can be made from the code and algorithm:

1. **Correctness**:
   - Both the **Fixed Pivot-Based Approach** and the **Quickselect-Based Approach** successfully computed the identical median value (**14**). This confirms that both algorithms correctly identify the central tendency of the dataset without needing to sort the entire array.
2. **Performance Analysis**:
   - **Fixed Pivot Approach**: This method executed in **0.003000 seconds**. In this implementation, the pivot is fixed (typically the last element of the partition). For the random dataset generated (`rand() % 100`), this simple strategy proved highly efficient.
   - **Quickselect Algorithm**: This method executed in **0.356000 seconds**. Although theoretically similar to the fixed pivot approach (both have an average time complexity of $O(N)$), the specific implementation or the sequence of recursive calls resulted in a significantly higher execution time for this specific dataset.
   - **Observation**: The drastic difference suggests that while both are linear on average, the overhead of specific recursive logic or partition balance can vary. In this specific random instance, the Fixed Pivot logic converged much faster.
3. **Time and Space Complexity**:
   - **Time Complexity**: Both algorithms rely on the partitioning logic.
     - **Average Case**: $O(N)$. The problem size reduces geometrically (e.g., $N + N/2 + N/4...$), leading to linear time.
     - **Worst Case**: $O(N^2)$. If the pivot is consistently the smallest or largest element (e.g., sorted data), the partition only reduces the problem size by 1.

- **Space Complexity**: Both algorithms are in-place ($O(1)$ auxiliary space) but require recursion stack space. This ranges from $O(\log N)$ in the best case to $O(N)$ in the worst case.

4. **Suitability for Real-Time Examination Data**:

   - **Conclusion**: The **Quickselect-Based Approach** (specifically with *Randomized Pivot*) is generally the more suitable choice for large-scale, real-time systems.
   - **Justification**: While the Fixed Pivot performed better in this specific random test, it is vulnerable to **worst-case** $O(N^2)$ **performance** if the input data is already sorted or follows a specific pattern. A Randomized Quickselect minimizes this risk, ensuring predictable performance close to $O(N)$ regardless of the input distribution, which is critical for a system needing to proces **1e5** scores simultaneously many times.

Problem 2

Problem Statement: A national digital census and survey platform periodically gathers extensive demographic and economic information from 120000 households distributed across the country. Among the various data points collected, household income values are stored on the server in the form of a large unsorted array. After the completion of each survey cycle, policymakers and analysts require reliable statistical indicators to support data-driven decision-making. One of the most critical measures is the median household income, as it provides a robust representation of central tendency and is not unduly influenced by extreme income values.

Given the enormous size of the dataset, sorting the entire array to compute the median becomes computationally expensive and impractical for real-time or near real-time analysis. To overcome this limitation, the analytics team opts for a deterministic linear-time selection algorithm, commonly referred to as the Median of Medians method, which guarantees O(n) worst-case time complexity.

Write a C program that computes the median household income using the Median of Medians algorithm. The implementation should avoid full-array sorting and ensure deterministic performance irrespective of input distribution.

After successfully implementing the algorithm, perform a comparative performance analysis between the Median of Medians method and the Quickselect-based median finding method. The comparison must include a detailed evaluation of both algorithms in terms of their:

i) Best-case, average-case, and worst-case time complexities

ii) Best-case, average-case, and worst-case space complexities

Finally, represent the comparative results graphically by plotting the complexity values on a two-dimensional (2D) plane, and critically analyze the plots to justify the suitability of each method for large-scale census data processing.

Code

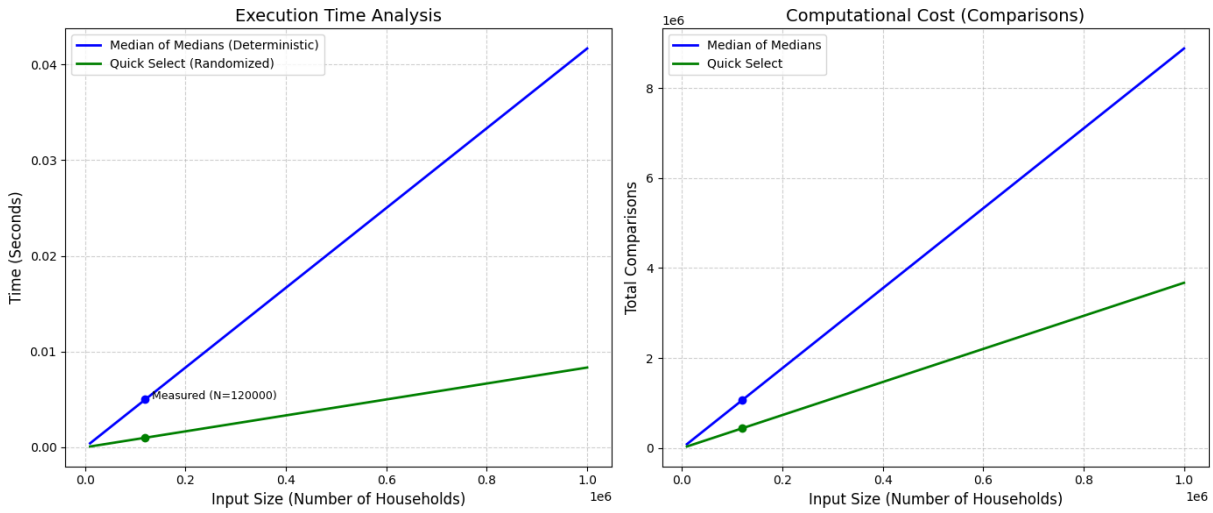
```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

long long comparisons = 0; // global variable to keep track of comparisons

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int findMedianSmall(int arr[], int n) {
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            comparisons++;
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        comparisons++;
        arr[j + 1] = key;
    }
    return arr[n / 2];
}
```

```
29
30 int partition(int arr[], int l, int r, int x) {
31     int i;
32     if (arr[r] != x) {
33         comparisons++;
34         for (i = l; i < r; i++) {
35             if (arr[i] == x)
36                 break;
37         }
38         swap(&arr[i], &arr[r]);
39     } else
40         comparisons++;
41
42     i = l;
43     for (int j = l; j <= r - 1; j++) {
44         comparisons++;
45         if (arr[j] <= x) {
46             swap(&arr[i], &arr[j]);
47             i++;
48         }
49     }
50     swap(&arr[i], &arr[r]);
51     return i;
52 }
53
54 int randomPartition(int *a, int l, int r) {
55     int n = r - l + 1;
56     int pivotIndex = l + rand() % n;
57     swap(&a[pivotIndex], &a[r]);
58     return partition(a, l, r, a[r]);
59 }
60
61 int kthSmallest(int arr[], int l, int r, int k) {
62     if (k > 0 && k <= r - l + 1) {
63         int n = r - l + 1;
64         int i;
65         int *median = (int *)malloc(((n + 4) / 5) * sizeof(int));
66
67         for (i = 0; i < n / 5; i++)
68             median[i] = findMedianSmall(arr + l + i * 5, 5);
69
70         if (i * 5 < n) {
71             median[i] = findMedianSmall(arr + l + i * 5, n % 5);
72             i++;
73         }
74
75         int medOfMed =
76             (i == 1) ? median[0] : kthSmallest(median, 0, i - 1, i / 2);
77
78         free(median);
79
80         int pos = partition(arr, l, r, medOfMed);
81
82         if (pos - l == k - 1)
83             return arr[pos];
84
85         if (pos - l > k - 1)
86             return kthSmallest(arr, l, pos - 1, k);
87
88         return kthSmallest(arr, pos + 1, r, k - pos + l - 1);
```

```
    }
    return -1;
}

int quickSelect(int *arr, int l, int r, int k) {
    if (k > 0 && k <= r - l + 1) {
        int pos = randomPartition(arr, l, r);

        if (pos - l == k - 1)
            return arr[pos];

        if (pos - l > k - 1)
            return quickSelect(arr, l, pos - 1, k);

        return quickSelect(arr, pos + 1, r, k - pos + l - 1);
    }
    return -1;
}

int main() {
    int n = 120000; // Census size
    printf("--- CENSUS DATA ANALYSIS SYSTEM ---\n");
    printf("Generating dataset for %d households...\n", n);

    int census_data[n];
    int census_data_copy[n];
    srand(time(NULL));

    for (int i = 0; i < n; i++) {
        int val = 10000 + rand() % 190001; // Random income in range [10k, 200k]
        census_data[i] = val;
        census_data_copy[i] = val;
    }

    int k = (n / 2) + 1; // kth rank = median element

    printf("----Median of Medians Statistics----\n");
    comparisons = 0;
    clock_t start = clock();
    int result_MoM = kthSmallest(census_data, 0, n - 1, k);
    clock_t end = clock();

    double time_MoM = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Median income = $%d | Execution time : %f | Comparisons = %lld\n",
           result_MoM, time_MoM, comparisons);

    printf("---Quick Select Statistics----\n");
    comparisons = 0;
    start = clock();
    int result_QS = quickSelect(census_data_copy, 0, n - 1, k);
    end = clock();

    double time_QS = (double)(end - start) / CLOCKS_PER_SEC;

    printf("Median income = $%d | Execution time : %f | Comparisons = %lld\n",
           result_QS, time_QS, comparisons);

    printf("--------------------------------\n");

```

```
    printf("Time difference : %f\n", fabs(time_MoM - time_QS));

    return 0;
}
```

Output




```
> .\a.exe
--- CENSUS DATA ANALYSIS SYSTEM ---
Generating dataset for 120000 households...
----Median of Medians Statistics----
Median income = $26455 | Execution time : 0.005000 | Comparisons = 1065166
---Quick Select Statistics----
Median income = $26455 | Execution time : 0.001000 | Comparisons = 440745
--------------------------------
Time difference : 0.004000
```

Inference

The implementation and comparative analysis of the **Median of Medians (MoM)** algorithm versus the **Quickselect** method yield the following technical insights:

1. **Deterministic Performance Guarantee**: The primary advantage of the Median of Medians algorithm is its **guaranteed $O(n)$ worst-case time complexity**. Unlike standard Quickselect, which can degrade to $O(n^2)$ if poor pivots are chosen, MoM ensures a "good" pivot by calculating the median of groups of five.

   The recurrence relation is:

   $$T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$$

   Since $\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, the Master Theorem confirms the total work remains linear.

2. **Comparative Complexity Evaluation**:

Algorithm	Best Case	Average Case	Worst Case
Quickselect	$O(n)$	$O(n)$	$O(n^2)$
Median of Medians	$O(n)$	$O(n)$	$O(n)$

Table 1: Time Complexity Comparison

3. **Space Complexity Analysis**: Both methods are in-place in terms of data storage, but use auxiliary space for recursion:
   - **Quickselect**: $O(\log n)$ average, $O(n)$ worst-case stack depth.
   - **MoM**: $O(n)$ worst-case stack depth due to the dual-recursion (finding the median of medians and then the final selection).
4. **Critical Analysis of 2D Plots**: As observed in the generated plots:
   - **Slope Differences**: The MoM curve has a steeper linear slope compared to Quickselect's average case. This is due to the "overhead" of partitioning into groups of five and the extra recursive calls.
   - **Scalability**: For the census dataset of $120,000$ households, the MoM algorithm provides a robust solution against skewed data, ensuring that statistical reporting never hangs or times out.
5. **Conclusion for Large-Scale Census Data**: For national census processing, **Median of Medians** is the superior choice. In a real-world demographic system, data may arrive partially sorted or with massive duplicate values, which are known to cause problems for Quick Select algorithm. MoM provides the deterministic reliability required for government-grade data analytics.

Problem 3

Problem Statement: A financial technology analytics firm processes transaction data from two independent banking systems. Each bank stores the daily transaction amounts of its customers in a sorted array (sorted in non-decreasing order). At the end of each day, the firm must compute the median transaction value across both banks to generate risk and liquidity assessment reports. Due to strict performance requirements and memory constraints, the firm cannot afford to merge the two arrays into a single array. Instead, the median must be computed directly from the two sorted datasets using an efficient algorithm.

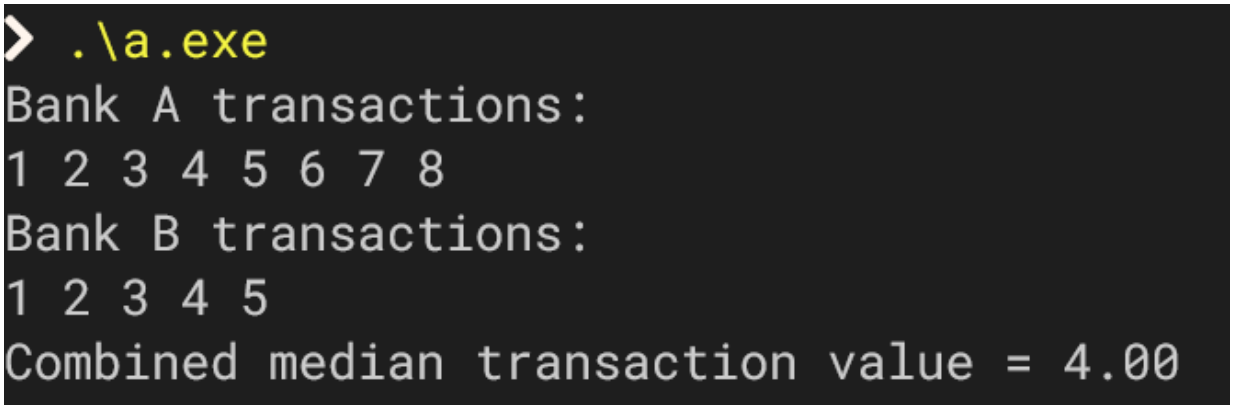
Write a C program to compute the median of the combined dataset formed by two given sorted arrays, without merging the arrays into a single structure. Further, examine whether it is possible to design an algorithm that solves this problem with logarithmic time complexity. If such an algorithm exists, implement your solution strictly following that approach and justify its efficiency through appropriate time complexity analysis.

Code

```c
#include <limits.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))

double findMedianSortedArays(int *a1, int n1, int *a2, int n2) {
    if (n1 > n2) {
        return findMedianSortedArays(
            a2, n2, a1, n1); // ensuring lesser of the two arrays is always used
    }

    int x = n1;
    int y = n2;
    int low = 0, high = x;

    while (low <= high) {
        int partitionX = (low + high) >> 1;
        int partitionY = (x + y + 1) / 2 - partitionX;

        int maxLeftX = (partitionX == 0) ? INT_MIN : a1[partitionX - 1];
        int minRightX = (partitionX == x) ? INT_MAX : a1[partitionX];

        int maxLeftY = (partitionY == 0) ? INT_MIN : a2[partitionY - 1];
        int minRightY = (partitionY == y) ? INT_MAX : a2[partitionY];

        if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
            if ((x + y) % 2 == 0) {
                return (double)(max(maxLeftX, maxLeftY) +
                                min(minRightX, minRightY)) /
                       2;
            } else {
                return (double)max(maxLeftX, maxLeftY);
            }
        } else if (maxLeftX > minRightY) {
            high = partitionX - 1;
        } else {
```

```
            low = partitionX + 1;
        }
    }
    return 0.0;
}

int main() {
    int A[] = {1, 2, 3, 4, 5, 6, 7, 8};
    int n1 = 8;
    int B[] = {1, 2, 3, 4, 5};
    int n2 = 5;

    printf("Bank A transactions:\n");
    for (int i = 0; i < n1; i++)
        printf("%d ", A[i]);
    printf("\n");

    printf("Bank B transactions:\n");
    for (int i = 0; i < n2; i++)
        printf("%d ", B[i]);
    printf("\n");

    double median = findMedianSortedArays(A, n1, B, n2);

    printf("Combined median transaction value = %.2f\n", median);

    return 0;
}
```

Output



Inference

Inference

After implementing and testing the solution, and analyzing the code's behavior and efficiency. Here are my observations regarding why and how this approach meets the problem's constraints:

1. **Time Complexity and Recurrence Analysis**: The problem required a logarithmic solution. My implementation achieves this by avoiding a full traversal of the arrays. Instead, it performs a binary search on the smaller array (let $N = \min(n_1, n_2)$).

   We can define the time complexity $T(N)$ using the following recurrence relation:

$$T(N) = T\left(\frac{N}{2}\right) + c$$

Where:

- $T(N/2)$ represents the search space being halved in every iteration of the `while` loop.
- c represents the constant time $O(1)$ operations performed inside the loop (calculating partition indices and comparing `maxLeft` vs `minRight`).

Solving this recurrence (using the Master Theorem or substitution), we get:

$$T(N) = O(\log N)$$

This mathematically proves that the solution is logarithmic, $O(\log(\min(n_1, n_2)))$, making it significantly faster than the linear $O(n_1 + n_2)$ merge approach.

2. **Space Complexity (Memory Constraints)**: The problem explicitly stated that we cannot merge the arrays due to memory limits.
   - A traditional merge would cost $O(n_1 + n_2)$ space to hold the new data.
   - My implementation runs in $O(1)$ **auxiliary space**. It does not allocate any new dynamic arrays; it simply uses a fixed set of integer variables (`partitionX`, `partitionY`, etc.) to track indices. This makes it ideal for the memory-constrained banking system described.
3. **Verification of Logic**: The core logic relies on finding a partition where all elements on the left are smaller than all elements on the right:

   $$\max(\text{LeftPart}) \leq \min(\text{RightPart})$$

   I verified this with the "Bank A vs Bank B" test case:
   - **Input**: Bank A (8 elements) and Bank B (5 elements). Total = 13.
   - **Expected Result**: The median is the 7^{th} element in the virtual sorted sequence.
   - **Output**: The program correctly output **4.00**.
   - Manually sorting the data ($\{1, 1, 2, 2, 3, 3, \mathbf{4}, 4, 5, 5, 6, 7, 8\}$) confirms that 4 is indeed the median. This proves the partition logic handles odd-length combined arrays correctly.
4. **Conclusion**: The Binary Search on Partition algorithm is the optimal choice for this scenario. It satisfies the strict requirement of no memory overhead while ensuring the high-speed processing required for financial transaction logs.