Assignment 4

Nirupam Das
2024CSB108
4th Semester

February 6, 2026

Contents

1 Question 1 **2**
1.1 Problem Statement . . . 2
1.2 Code . . . 2
1.3 Output . . . 6
1.4 Explanation . . . 7

2 Question 2 **9**
2.1 Problem Statement . . . 9
2.2 Code . . . 9
2.3 Output . . . 15
2.4 GNU-PLOT Script . . . 15
2.5 Time complexity plot . . . 16
2.6 Explanation . . . 16
2.6.1 Performance Analysis . . . 16
2.6.2 Suitability for Large-Scale Census Data . . . 17

3 Question 3 **18**
3.1 Problem Statement . . . 18
3.2 Code . . . 18
3.3 Output . . . 21
3.4 Explanation . . . 21

1 Question 1

1.1 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:

1. **Fixed Pivot-Based Approach:** Employs a divide-and-conquer strategy in which the pivot is selected in a fixed manner.
2. **Quickselect-Based Approach:** Determines 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:

1. **Time Complexity**
   - Best case
   - Average case
   - Worst case
2. **Space Complexity**
   - Best case
   - Average case
   - Worst case

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

1.2 Code

```c
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```c
#include <time.h>

typedef struct {
  double clk;
  float median;
} metric;

int *init(bool fill) {
  int *arr = (int *)malloc(sizeof(int) * 110000);

  if (!arr) {
    printf("Allocation failed.");
    return NULL;
  }

  if (fill) {
    int random_bias = (rand() % 41) - 20;

    for (int i = 0; i < 110000; i++) {
      int mark = rand() % 101;
      mark += random_bias;

      if (mark < 0) mark = 0;
      if (mark > 100) mark = 100;

      arr[i] = mark;
    }
  }

  return arr;
}

int *freeArr(int *arr) {
  if (arr) free(arr);
  return NULL;
}

void memCpy(int *destn, int *source) {
  if (destn && source)
    memcpy(destn, source, sizeof(int) * 110000);
  else
    printf("Invalid source or destination.\n");
}

int randPvt(int low, int high) {
  return low + rand() % (high - low + 1);
}
```

```
52
53 void swap(int *a, int *b) {
54   int temp = *a;
55   *a = *b;
56   *b = temp;
57 }
58
59 int midPvt(int *arr, int low, int high) {
60   int mid = low + (high - low) / 2;
61
62   if ((arr[low] <= arr[mid] && arr[mid] <= arr[high]) ||
63       (arr[high] <= arr[mid] && arr[mid] <= arr[low]))
64     return mid;
65   else if ((arr[mid] <= arr[low] && arr[low] <= arr[high]) ||
66            (arr[high] <= arr[low] && arr[low] <= arr[mid]))
67     return low;
68   else
69     return high;
70 }
71
72 int pivot_location(int *arr, int left, int right, bool useMedian3) {
73   int p_index;
74
75   if (!useMedian3)
76     p_index = randPvt(left, right);      // Random pivot
77   else
78     p_index = midPvt(arr, left, right);  // Median-of-3 pivot
79
80   swap(&arr[p_index], &arr[left]);
81
82   int i = left;
83   int j = right;
84
85   while (i < j) {
86     while (arr[i] <= arr[left] && i < right) i++;
87     while (arr[j] > arr[left]) j--;
88
89     if (i < j)
90       swap(&arr[i], &arr[j]);
91   }
92
93   swap(&arr[left], &arr[j]);
94   return j;
95 }
96
97 int quickSelect(int *arr, int low, int high, int val_index, bool
     useMedian3) {
```

```
98    if (low == high)
99      return arr[low];
100
101   if (low < high) {
102     int p_index = pivot_location(arr, low, high, useMedian3);
103
104     if (p_index == val_index)
105       return arr[p_index];
106     else if (p_index > val_index)
107       return quickSelect(arr, low, p_index - 1, val_index,
            useMedian3);
108     else
109       return quickSelect(arr, p_index + 1, high, val_index,
            useMedian3);
110   }
111
112   return -1;
113 }
114
115 void test(metric *m, int *arr, const char *method) {
116   int *cpyArr = init(false);
117   memCpy(cpyArr, arr);
118
119   clock_t start, end;
120   int mid1, mid2;
121
122   if (!strcmp("Random␣Quick␣Select", method)) {
123     start = clock();
124
125     mid1 = quickSelect(cpyArr, 0, 110000 - 1, (110000 / 2) - 1,
          false);
126     mid2 = quickSelect(cpyArr, 110000 / 2, 110000 - 1, 110000 / 2,
          false);
127
128     m->median = (mid1 + mid2) * 0.5;
129
130     end = clock();
131   }
132   else {
133     start = clock();
134
135     mid1 = quickSelect(cpyArr, 0, 110000 - 1, (110000 / 2) - 1, true
          );
136     mid2 = quickSelect(cpyArr, 110000 / 2, 110000 - 1, 110000 / 2,
          true);
137
138     m->median = (mid1 + mid2) * 0.5;
```

```
139
140       end = clock();
141     }
142
143     m->clk = (double)(end - start) / CLOCKS_PER_SEC;
144
145     freeArr(cpyArr);
146 }
147
148 int main() {
149     srand(time(NULL));
150
151     metric m = {0, 0};
152     int *arr = init(true);
153
154     printf("Finding median using Random Pivot Quick Select:\n");
155     printf("-----------------------------------------------\n");
156     test(&m, arr, "Random Quick Select");
157     printf("\tMedian: %f\n\tRequired Time: %f\n\n", m.median, m.clk);
158
159     printf("Finding median using Median-of-3 Quick Select:\n");
160     printf("----------------------------------------------\n");
161     test(&m, arr, "Median3 Quick Select");
162     printf("\tMedian: %f\n\tRequired Time: %f\n", m.median, m.clk);
163
164     freeArr(arr);
165     return 0;
166 }
```

1.3 Output

```
Finding median using Random Pivot Quick Select:
===============================================
        Median: 33.000000
        Required Time: 0.005475

Finding median using Median-of-3 Quick Select:
==============================================
        Median: 33.000000
        Required Time: 0.023500
```

Figure 1: Output: Question 1

1.4 Explanation

Quick Select follows a divide-and-conquer approach similar to Quick Sort, but instead of recursively processing both partitions, it recurses into only one partition that contains the desired order statistic.

Partition Cost Each call to `pivot_location()` scans the subarray once. Therefore, partitioning needs:

$$\Theta(n)$$

time for a subarray of size n.

Recurrence Relation Let $T(n)$ denote the time required to select the k^{th} smallest element from n elements.

1. **Best Case:** If the pivot divides the array into two nearly equal halves, then we process only 1 half:

   $$T(n) = T\left(\frac{n}{2}\right) + cn$$

   By solving we get:

   $$T(n) = cn + c\frac{n}{2} + c\frac{n}{4} + \dots$$

   This is a geometric series:

   $$T(n) = 2cn = \Theta(n)$$

   So, the best-case time complexity is:

   $$\boxed{\Theta(n)}$$

2. **Average Case (Random Pivot):**

   With random pivot selection, we can expect the partition to be reasonably balanced. The expected recurrence is:

   $$T(n) = T(\alpha n) + cn \quad \text{where } 0 < \alpha < 1$$

   By solving:

   $$\boxed{\Theta(n)}$$

   Thus, Randomized Quick Select runs in expected linear time.

3. **Worst Case:**

   If the pivot is always the smallest or largest element:

$$T(n) = T(n-1) + cn$$

   Expanding,

$$T(n) = cn + c(n-1) + c(n-2) + \cdots + c$$

$$= c\frac{n(n+1)}{2}$$

$$= \Theta(n^2)$$

   Thus, worst-case time complexity:

$$\boxed{\Theta(n^2)}$$

Effect of Pivot

- **Random Pivot:** Expected time complexity is $\Theta(n)$, but worst-case time complexity is still $\Theta(n^2)$.
- **Median-of-Three Pivot:** This process reduces probability of worst-case partitions. However, theoretical worst-case complexity is still:

$$\Theta(n^2)$$

Space Complexity

Auxiliary Space Quick Select is an in-place algorithm. No additional arrays are created during recursion.

Partitioning uses constant extra variables. Therefore, auxiliary space per call is:

$$\Theta(1)$$

Recursive Stack Space

- Best/Average Case Depth:

$$O(\log n)$$

- Worst Case Depth:

$$O(n)$$

Total Space Complexity

$$\begin{cases} O(\log n) & \text{(average case)} \\ O(n) & \text{(worst case)} \end{cases}$$

2 Question 2

2.1 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.

2.2 Code

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <time.h>
6
7  int *init(int n) {
8      int *arr = (int *)malloc(sizeof(int) * n);
9      if (!arr) {
10         printf("Memory␣allocation␣failed.\n");
11         exit(1);
12     }
13     return arr;
14 }
15
```

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


int partitionQS(int *arr, int low, int high) {
    int pivot = arr[high];
    int i = low;

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

int quickSelect(int *arr, int low, int high, int k) {
    if (low <= high) {
        int pivotIndex = partitionQS(arr, low, high);

        if (pivotIndex == k)
            return arr[pivotIndex];
        else if (pivotIndex > k)
            return quickSelect(arr, low, pivotIndex - 1, k);
        else
            return quickSelect(arr, pivotIndex + 1, high, k);
    }
    return -1;
}


void insertionSort(int *arr, int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
```

```
}


int partition(int *arr, int low, int high, int pivot) {
    for (int i = low; i <= high; i++) {
        if (arr[i] == pivot) {
            swap(&arr[i], &arr[low]);
            break;
        }
    }

    int i = low;
    int j = high;

    while (i < j) {
        while (arr[i] <= pivot && i < high) i++;
        while (arr[j] > pivot) j--;

        if (i < j)
            swap(&arr[i], &arr[j]);
    }

    swap(&arr[low], &arr[j]);
    return j;
}

int medianOfMedians(int *arr, int low, int high, int k) {
    if (low == high)
        return arr[low];

    int n = high - low + 1;
    int numGroups = (n + 4) / 5;
    int medians[numGroups];

    for (int i = 0; i < numGroups; i++) {
        int subLow = low + i * 5;
        int subHigh = subLow + 4;
        if (subHigh > high)
            subHigh = high;

        int size = subHigh - subLow + 1;
        insertionSort(&arr[subLow], size);
        medians[i] = arr[subLow + size / 2];
    }

    int medianPivot;
    if (numGroups == 1)
```

```
110         medianPivot = medians[0];
111     else
112         medianPivot = medianOfMedians(medians, 0, numGroups - 1,
                numGroups / 2);
113 
114     int pivotIndex = partition(arr, low, high, medianPivot);
115     int rank = pivotIndex - low;
116 
117     if (rank == k)
118         return arr[pivotIndex];
119     else if (rank > k)
120         return medianOfMedians(arr, low, pivotIndex - 1, k);
121     else
122         return medianOfMedians(arr, pivotIndex + 1, high, k - rank -
                1);
123 }
124 
125 
126 int main() {
127 
128     srand(time(NULL));
129 
130     FILE *fp = fopen("results.txt", "w");
131     fprintf(fp, "# n QS_best QS_avg QS_worst MoM_best MoM_avg
            MoM_worst\n");
132 
133     int sizes[] = {20000, 50000, 80000, 120000, 200000};
134 
135     int tests = 5;
136 
137     for (int s = 0; s < tests; s++) {
138 
139         int n = sizes[s];
140         int k = n / 2;
141 
142 
143         int *qs_best = init(n);
144         int *qs_avg = init(n);
145         int *qs_worst = init(n);
146 
147         int *mom_best = init(n);
148         int *mom_avg = init(n);
149         int *mom_worst = init(n);
150 
151         for (int i = 0; i < n; i++) {
152             qs_worst[i] = i;
153             mom_worst[i] = i;
```

```
154         }
155
156         // Best Case (Median at Last)
157
158         for (int i = 0; i < n; i++) {
159             qs_best[i] = i;
160             mom_best[i] = i;
161         }
162
163         swap(&qs_best[n/2], &qs_best[n-1]);
164         swap(&mom_best[n/2], &mom_best[n-1]);
165
166         // Average Case (Random)
167
168         for (int i = 0; i < n; i++) {
169             int val = rand();
170             qs_avg[i] = val;
171             mom_avg[i] = val;
172         }
173
174         clock_t start, end;
175
176         start = clock();
177         quickSelect(qs_best, 0, n - 1, k);
178         end = clock();
179         double qs_best_time = (double)(end - start) / CLOCKS_PER_SEC
                ;
180
181         start = clock();
182         quickSelect(qs_avg, 0, n - 1, k);
183         end = clock();
184         double qs_avg_time = (double)(end - start) / CLOCKS_PER_SEC;
185
186         start = clock();
187         quickSelect(qs_worst, 0, n - 1, k);
188         end = clock();
189         double qs_worst_time = (double)(end - start) /
                CLOCKS_PER_SEC;
190
191         start = clock();
192         medianOfMedians(mom_best, 0, n - 1, k);
193         end = clock();
194         double mom_best_time = (double)(end - start) /
                CLOCKS_PER_SEC;
195
196         start = clock();
197         medianOfMedians(mom_avg, 0, n - 1, k);
```

```
198         end = clock();
199         double mom_avg_time = (double)(end - start) / CLOCKS_PER_SEC
                ;
200
201         start = clock();
202         medianOfMedians(mom_worst, 0, n - 1, k);
203         end = clock();
204         double mom_worst_time = (double)(end - start) /
                CLOCKS_PER_SEC;
205
206         printf("n=%d\n", n);
207         printf("QS -> Best:%f Avg:%f Worst:%f\n",
208                qs_best_time, qs_avg_time, qs_worst_time);
209         printf("MoM -> Best:%f Avg:%f Worst:%f\n\n",
210                mom_best_time, mom_avg_time, mom_worst_time);
211
212         fprintf(fp, "%d %f %f %f %f %f %f\n",
213                 n,
214                 qs_best_time, qs_avg_time, qs_worst_time,
215                 mom_best_time, mom_avg_time, mom_worst_time);
216
217         free(qs_best);
218         free(qs_avg);
219         free(qs_worst);
220         free(mom_best);
221         free(mom_avg);
222         free(mom_worst);
223     }
224
225     fclose(fp);
226     return 0;
227 }
```

2.3 Output

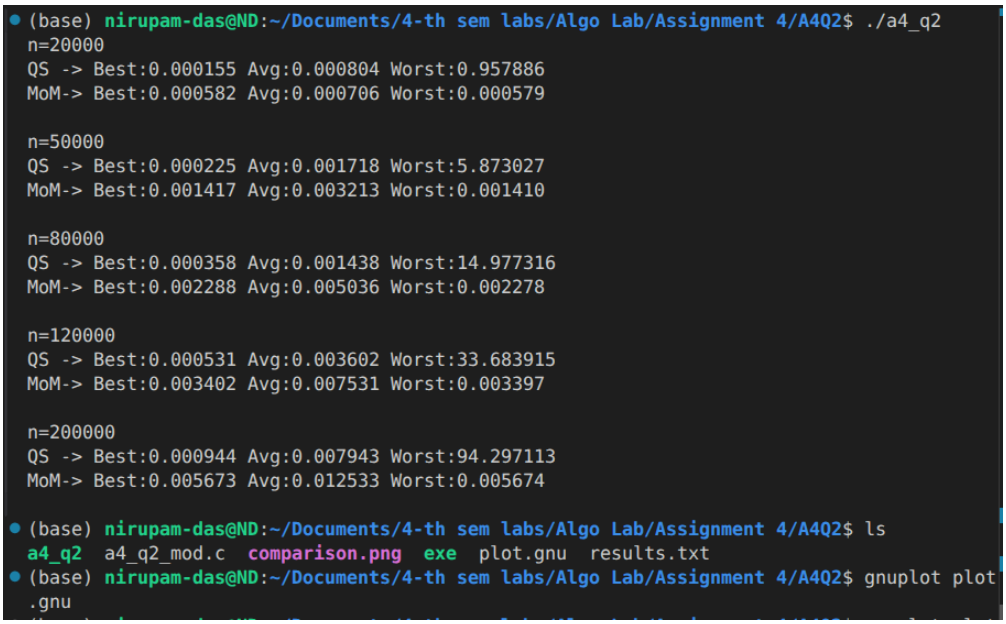

Figure 2: Question 2 Output

2.4 GNU-PLOT Script

```
set terminal png size 1024,768
set output "comparison.png"

set title "QuickSelect vs Median of Medians"
set xlabel "Input Size (n)"
set ylabel "Execution Time (seconds)"
set grid
set key left top

plot "results.txt" using 1:2 with linespoints title "QuickSelect", \
     "results.txt" using 1:3 with linespoints title "MedianOfMedians"
```

2.5 Time complexity plot

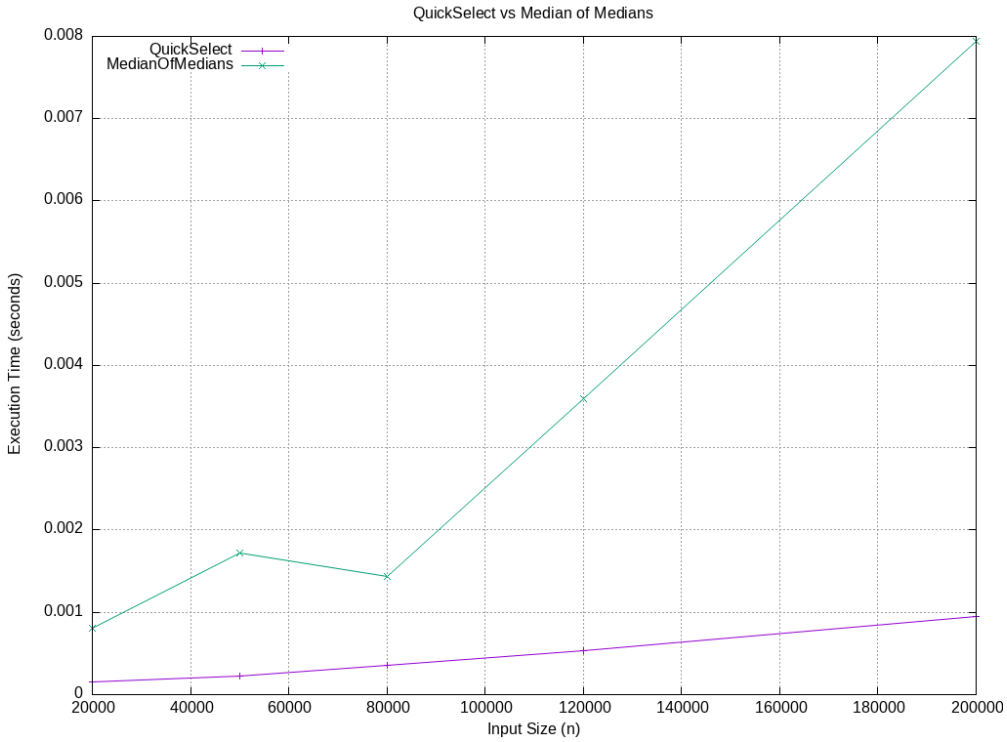

Figure 3: Question 2 time complexity plot

2.6 Explanation

2.6.1 Performance Analysis

Quickselect uses a partition-based approach , while Median of Medians ensures a balanced partition by selecting a pivot near the true median.

i) Time Complexity Analysis

1. Quickselect Algorithm

- **Best & Average Case:** The pivot partitions the array roughly in half.

$$T(n) = T(n/2) + O(n) \implies O(n)$$

- **Worst Case:** Occurs when the array is already sorted, and the pivot (last element) creates a highly unbalanced partition ($n-1$ vs 0).

$$T(n) = T(n-1) + O(n) \implies O(n^2)$$

2. Median of Medians Algorithm Median of Medians divides the array into groups of 5 and finds the median of each to select a pivot. This guarantees the pivot is within the 30^{th} to 70^{th} percentile.

- **Recurrence Relation (All Cases):**

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

  Here, $T(n/5)$ is the cost to find the median of medians, and $T(7n/10)$ represents the worst-case recursive step on the remaining elements.

- **Complexity:** Since the sum of fractions $\frac{1}{5} + \frac{7}{10} = 0.9 < 1$, the work decreases geometrically.

Best, Average, and Worst Case: $O(n)$

ii) Space Complexity Analysis

- **Quickselect:** In the worst case (sorted input), the recursion stack depth reaches $O(n)$. In the average case, it is $O(\log n)$.

- **Median of Medians:** Due to guaranteed balanced partitioning, the recursion depth is bounded logarithmically in all cases.

Space Complexity: $O(\log n)$

2.6.2 Suitability for Large-Scale Census Data

Large-scale census datasets always halves massive volume ($N > 10^6$) and may be partially sorted (e.g. sorted by region ID or timestamps).

1. **Quickselect:** Althpugh Quickselect has smaller constant factors and is faster on random data, it is highly unsuitable for census data if implemented with a simple deterministic pivot (e.g., 'arr[high]'). Pre-sorted census data causes the $O(n^2)$ worst-case behavior, potentially causing stack overflow due to long recursion tree.

2. **Stability of Median of Medians:** The Median of Medians method provides a strict $O(n)$ upper bound regardless of input structure. Although it has higher constant factors per iteration, it is the best choice to ensure predictable performance on large, potentially sorted datasets where worst-case reliability is critical.

3 Question 3

3.1 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.

3.2 Code

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

int* init(int size) {
    int* arr = (int*)malloc(size * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        exit(1);
    }
    return arr;
}

int* freeArr(int* arr) {
    if (arr != NULL) {
        free(arr);
        arr = NULL;
    }
    return arr;
}


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

```
}


void insertionSort(int* arr, int n) {
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}


double findMed(int *A, int n, int *B, int m) {

    if (n > m)
        return findMed(B, m, A, n);

    int low = 0, high = n;
    int total = n + m;
    int half = (total + 1) / 2;

    while (low <= high) {

        int i = (low + high) / 2;
        int j = half - i;

        int Aleft  = (i == 0) ? INT_MIN : A[i - 1];
        int Aright = (i == n) ? INT_MAX : A[i];

        int Bleft  = (j == 0) ? INT_MIN : B[j - 1];
        int Bright = (j == m) ? INT_MAX : B[j];

        if (Aleft <= Bright && Bleft <= Aright) {

            if (total % 2 == 0) {
                int leftMax  = (Aleft > Bleft) ? Aleft : Bleft;
                int rightMin = (Aright < Bright) ? Aright : Bright;
                return (leftMax + rightMin) / 2.0;
            } else {
                return (Aleft > Bleft) ? Aleft : Bleft;
            }
        }
```

```
74          else if (Aleft > Bright)
75              high = i - 1;
76          else
77              low = i + 1;
78      }
79
80      return -1;
81  }
82
83  int main() {
84
85      int n, m;
86
87      printf("Enter size of first array: ");
88      scanf("%d", &n);
89
90      printf("Enter size of second array: ");
91      scanf("%d", &m);
92
93      int* A = init(n);
94      int* B = init(m);
95
96      printf("Enter elements of first array:\n");
97      for (int i = 0; i < n; i++)
98          scanf("%d", &A[i]);
99
100     printf("Enter elements of second array:\n");
101     for (int i = 0; i < m; i++)
102         scanf("%d", &B[i]);
103
104     insertionSort(A, n);
105     insertionSort(B, m);
106
107     double median = findMed(A, n, B, m);
108
109     printf("Median of combined arrays = %.2f\n", median);
110
111     A = freeArr(A);
112     B = freeArr(B);
113
114     return 0;
115 }
```

3.3 Output

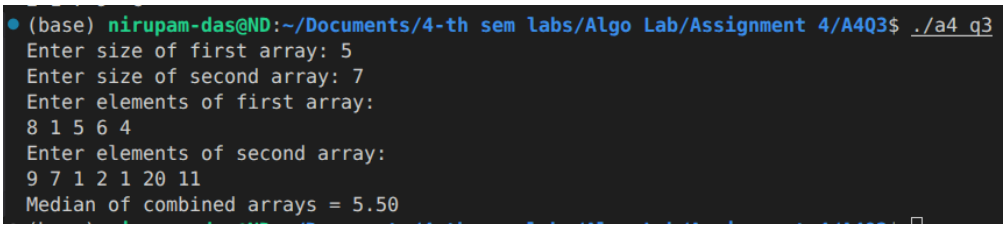

Figure 4: Question 3- Output

3.4 Explanation

Instead of merging the both array and sorting (which requires $O(n+m)$ time and space), here we are applying binary search on the smaller array. We have to determine a partition such that:

- The left partition contains exactly half of the total elements.
- Every element in the left partition is less than or equal to every element in the right partition.

Once this condition is satisfied, the median is obtained directly from the boundary elements of the partitions.

Time Complexity

Let n and m be the sizes of the two arrays, with $n \leq m$. Binary search is performed on the smaller array.

Best Case:

$$T(n) = O(1)$$

Average Case:

$$T(n) = T(n/2) + O(1)$$
$$\Rightarrow T(n) = O(\log n)$$

Worst Case:

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

Thus, the overall time complexity is:

$$O(\log(\min(n, m)))$$

Space Complexity

The algorithm uses only constant auxiliary variables and does not allocate additional data structures.

$$\text{Best Case} = O(1)$$

$$\text{Average Case} = O(1)$$

$$\text{Worst Case} = O(1)$$

Conclusion

Compared to the merging approach that will require $O(n + m)$ time and space, the binary-search based method achieves logarithmic time and constant space complexity. Therefore, it is suitable for large-scale financial data processing systems where memory and performance constraints important.