Assignment 4

Nirupam Das
2024CSB108
4th Semester

February 6, 2026

Contents

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

```
1  #include <stdbool.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #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);
}
```

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

int midPvt(int *arr, int low, int high) {
  int mid = low + (high - low) / 2;

  if ((arr[low] <= arr[mid] && arr[mid] <= arr[high]) ||
      (arr[high] <= arr[mid] && arr[mid] <= arr[low]))
    return mid;
  else if ((arr[mid] <= arr[low] && arr[low] <= arr[high]) ||
           (arr[high] <= arr[low] && arr[low] <= arr[mid]))
    return low;
  else
    return high;
}

int pivot_location(int *arr, int left, int right, bool useMedian3) {
  int p_index;

  if (!useMedian3)
    p_index = randPvt(left, right);       // Random pivot
  else
    p_index = midPvt(arr, left, right);  // Median-of-3 pivot

  swap(&arr[p_index], &arr[left]);

  int i = left;
  int j = right;

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

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

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

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

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 kth 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:

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

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

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

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

```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;
    }
```

```
63  }
64
65
66  int partition(int *arr, int low, int high, int pivot) {
67      for (int i = low; i <= high; i++) {
68          if (arr[i] == pivot) {
69              swap(&arr[i], &arr[low]);
70              break;
71          }
72      }
73
74      int i = low;
75      int j = high;
76
77      while (i < j) {
78          while (arr[i] <= pivot && i < high) i++;
79          while (arr[j] > pivot) j--;
80
81          if (i < j)
82              swap(&arr[i], &arr[j]);
83      }
84
85      swap(&arr[low], &arr[j]);
86      return j;
87  }
88
89  int medianOfMedians(int *arr, int low, int high, int k) {
90      if (low == high)
91          return arr[low];
92
93      int n = high - low + 1;
94      int numGroups = (n + 4) / 5;
95      int medians[numGroups];
96
97      for (int i = 0; i < numGroups; i++) {
98          int subLow = low + i * 5;
99          int subHigh = subLow + 4;
100         if (subHigh > high)
101             subHigh = high;
102
103         int size = subHigh - subLow + 1;
104         insertionSort(&arr[subLow], size);
105         medians[i] = arr[subLow + size / 2];
106     }
107
108     int medianPivot;
109     if (numGroups == 1)
```

```c
                medianPivot = medians[0];
        else
                medianPivot = medianOfMedians(medians, 0, numGroups - 1,
                    numGroups / 2);

        int pivotIndex = partition(arr, low, high, medianPivot);
        int rank = pivotIndex - low;

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


int main() {

        srand(time(NULL));

        FILE *fp = fopen("results.txt", "w");
        fprintf(fp, "#_n_QS_best_QS_avg_QS_worst_MoM_best_MoM_avg_
            MoM_worst\n");

        int sizes[] = {20000, 50000, 80000, 120000, 200000};

        int tests = 5;

        for (int s = 0; s < tests; s++) {

                int n = sizes[s];
                int k = n / 2;


                int *qs_best = init(n);
                int *qs_avg = init(n);
                int *qs_worst = init(n);

                int *mom_best = init(n);
                int *mom_avg = init(n);
                int *mom_worst = init(n);

                for (int i = 0; i < n; i++) {
                        qs_worst[i] = i;
                        mom_worst[i] = i;
```

12

```c
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"
```
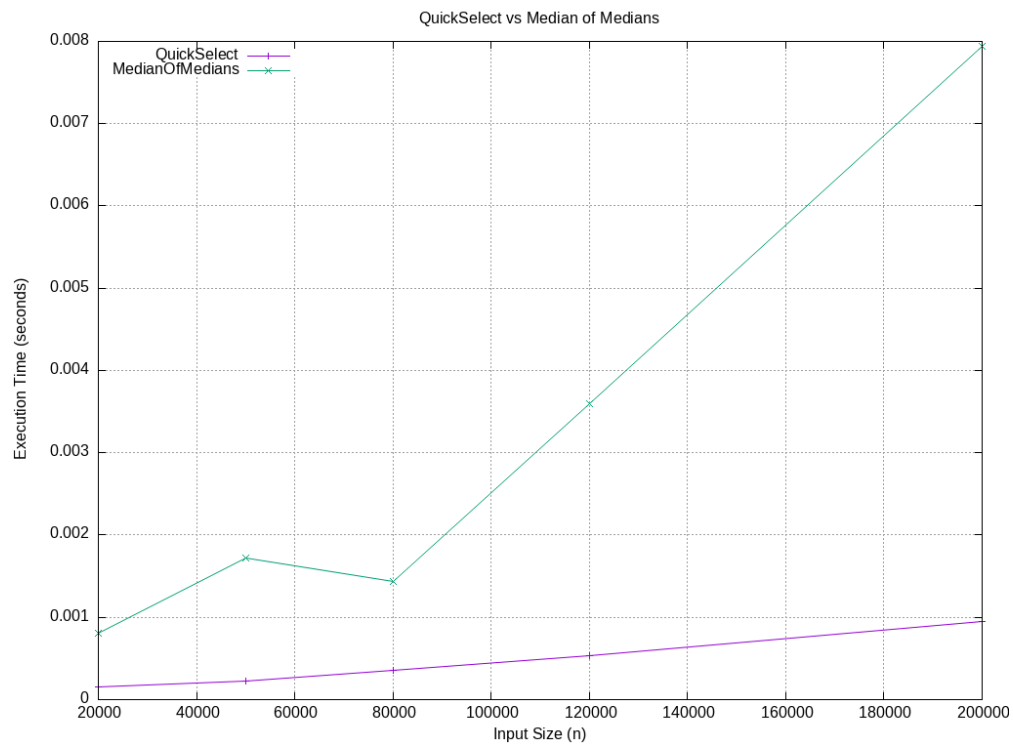
2.5 Time complexity plot

Figure 3: Question 2 time complexity plot

2.6 Explanation