Design and Analysis of Algorithms Activity

Group Members:

1) Harshdeep (590014572)

2) Shiva Tyagi (590012350)

3) Lakshay Manchanda (590011784)

4) Taraksh Goyal (590011425)

5) Aakshat Garg (590012092)

6) Ayush Chauhan (590011658)

1. Problem Statement & Significance

Median of Medians Selection (Divide and Conquer)

The **Median of Medians** is a selection algorithm that finds the *k-th smallest element* (and therefore the median) in an unsorted list in **linear time, O(n)**.

Unlike naive pivot selection methods—such as always picking the first element or choosing randomly—this algorithm carefully chooses a pivot using the “median of medians” strategy.

- With poor pivot choices, selection algorithms can degrade to **quadratic time $O(n^2)$** in the worst case, or **O(n log n)** on average.
- The Median of Medians algorithm guarantees a **balanced pivot** at every step, ensuring **worst-case O(n) performance**.

0	1	2	3	4	5	6	7	8	9	10
10	1	67	20	56	8	43	90	54	34	0

Median

0 1 8 10 20 **34** 43 54 56 67 90

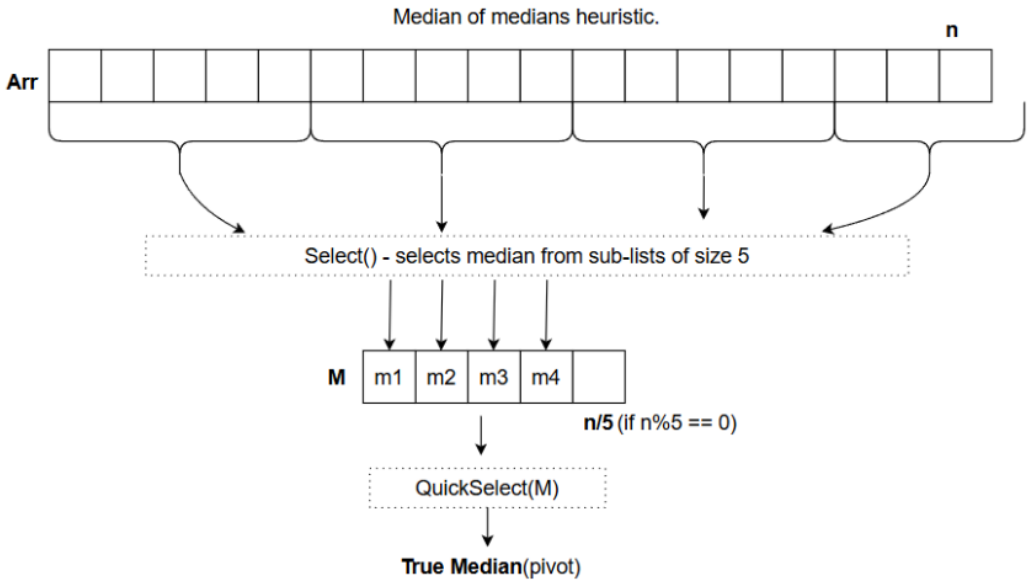
Sort the array

O(N log(N))

Get the median

To reduce time complexity, the **Median of Medians selection algorithm** works by **recursively computing an approximate median** from the medians of small subgroups. This approximate median serves as a **robust pivot** in the divide-and-conquer selection process, ensuring balanced partitions and achieving **linear-time complexity** in the worst case.

This strategy provides a guaranteed linear worst-case time complexity.



Steps for Median of Medians Algorithm

1. **Divide into groups:**
   Split the unsorted array into groups of 5 elements each.
   a. The last group may contain fewer than 5 elements (but at least 1).
2. **Find medians of groups:**
   a. Sort each group individually.
   b. Select the median element from each group.
   c. Collect these medians into a new array.
3. **Select pivot (median of medians):**
   a. If the number of medians is more than 5, recursively apply the same process on this new array of medians.
   b. Otherwise, sort the medians and choose their median.
   c. This chosen element is the **pivot**.
4. **Partition around pivot:**
   a. Partition the original array into three parts:
      i. Elements less than pivot (Left).
      ii. Elements equal to pivot.
      iii. Elements greater than pivot (Right).
   b. The pivot's final index is now known.
5. **Recurse on correct side:**
   a. If the pivot's index equals the desired index (e.g., middle element for median), return the pivot.
   b. If the pivot's index is greater than the desired index, recursively apply the algorithm to the **left subarray**.
   c. If the pivot's index is smaller than the desired index, recursively apply the algorithm to the **right subarray**.

2. Pseudocode: Brute Force Method:

```
Algorithm BruteForceMedian(arr, k):
1. Sort the array using any sorting algorithm (e.g., MergeSort)
2. Return arr[k-1]
```

Pseudocode: Median of Medians Selection

```
Algo SELECT(A, k):
    # A = array of n elements
    # k = index of k-th smallest element (0-based)

    n ← length(A)

    IF n ≤ 5 THEN
        SORT(A)
        RETURN A[k]
    END IF

    # Step 1: Divide into groups of 5                    ~{O(n)}
    groups ← divide A into groups of size at most 5

    # Step 2: Find medians of each group                 ~{O(n)}
    medians ← empty list
    FOR each group g in groups DO
        SORT(g)
        medians.append( g[ floor(length(g)/2) ] )
    END FOR

    # Step 3: Select pivot (median of medians)           ~{n/5}
    pivot ← SELECT(medians, floor(length(medians)/2))

    # Step 4: Partition around pivot                     ~{O(n)}
    L ← [ x ∈ A such that x < pivot ]
    E ← [ x ∈ A such that x = pivot ]
    G ← [ x ∈ A such that x > pivot ]

    # Step 5: Recurse into correct subarray              ~{<=7n/10}
    IF k < length(L) THEN
        RETURN SELECT(L, k)
    ELSE IF k < length(L) + length(E) THEN
        RETURN pivot
    ELSE
        RETURN SELECT(G, k - length(L) - length(E))
    END IF
END FUNCTION
```

3. Analysis of Pseudocode

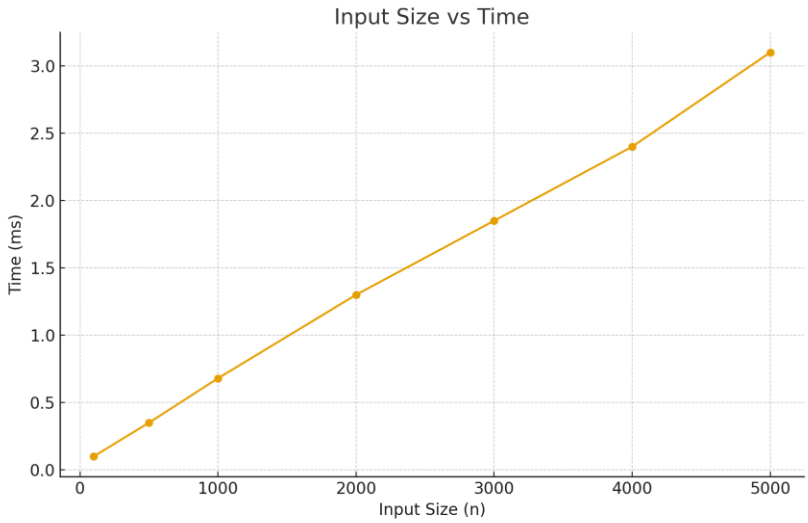
a) Recurrence Relations:

1. Brute Force: $T(n) = O(n \log n)$
2. Median of Medians: $T(n) <= T(n/5) + T(7n/10) + O(n)$

b) Solved Time Complexities:

- Brute Force: $O(n \log n)$
- Median of Medians: $O(n)$

c) Graph for Input Size vs Time:



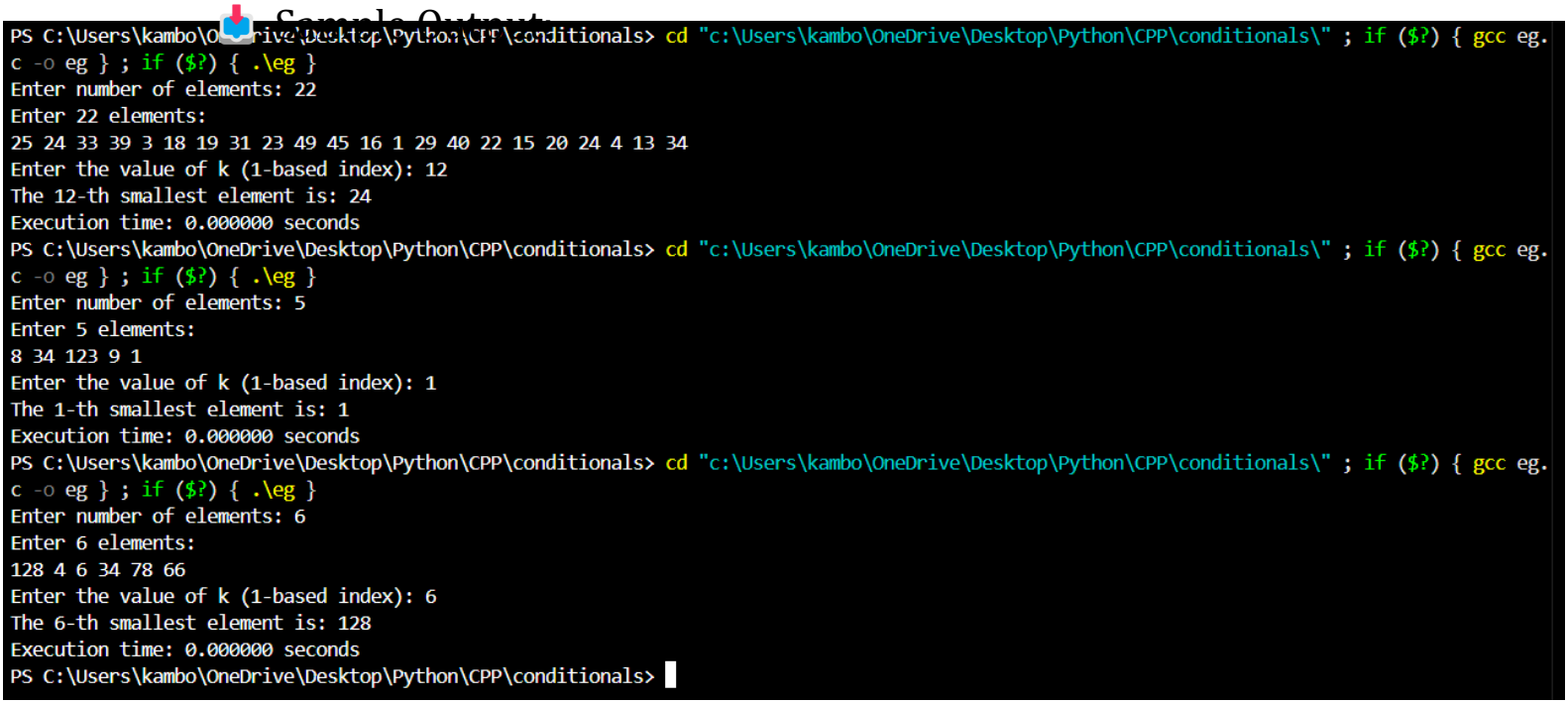
4. C Code for Median of Medians Algorithm (User Input Version)

```
#include <stdio.h>
#include <stdlib.h>
#include<time.h>
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 findMedian(int arr[], int n) {
   insertionSort(arr, n);
   return arr[n / 2];
}

int partition(int arr[], int n, int pivot) {
   int temp[n], i, j = 0;
   for (i = 0; i < n; i++)
```

```
    if (arr[i] < pivot) temp[j++] = arr[i];
  int mid = j;
  temp[j++] = pivot;
  for (i = 0; i < n; i++)
    if (arr[i] > pivot) temp[j++] = arr[i];
  for (i = 0; i < n; i++)
    arr[i] = temp[i];
  return mid;
}

int selectKth(int arr[], int n, int k) {
  if (n <= 5) {
    insertionSort(arr, n);
    return arr[k];
  }

  int groups = (n + 4) / 5;
  int medians[groups];
  for (int i = 0; i < groups; i++) {
    int start = i * 5;
    int len = (start + 5 > n) ? (n - start) : 5;
    medians[i] = findMedian(arr + start, len);
  }

  int pivot = selectKth(medians, groups, groups / 2);
  int pos = partition(arr, n, pivot);

  if (k < pos)
    return selectKth(arr, pos, k);
  else if (k > pos)
    return selectKth(arr + pos + 1, n - pos - 1, k - pos - 1);
  else
    return pivot;
}

int main() {
  int n, k;
  clock_t start, end;
  double cpu_time_used;


  printf("Enter number of elements: ");
  scanf("%d", &n);
```

```
    int arr[n];
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    printf("Enter the value of k (1-based index): ");
    scanf("%d", &k);

    if (k < 1 || k > n) {
        printf("Invalid value of k.\n");
        return 1;
    }


    start=clock();
    int result = selectKth(arr, n, k - 1);
    end=clock();

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


    printf("The %d-th smallest element is: %d\n", k, result);
    printf("Execution time: %f seconds\n", cpu_time_used);


    return 0;
}
```

Sample Output



Conclusion:

~ Advantages

- Guarantees **worst-case linear time complexity** $O(n)O(n)$O(n).
- Provides a **robust pivot selection** compared to random or naive methods.
- Ensures **balanced partitions**, avoiding performance degradation.
- Works well for theoretical analysis and critical systems where predictability is important.

~ Disadvantages

- Implementation is **more complex** than randomized quickselect.
- Higher **constant factors** make it slower in practice for small/medium inputs.
- Sorting small groups introduces **extra overhead** compared to randomized pivoting.

~ Applications

- Used in **selection problems** (finding k-th smallest/largest elements).
- Computing the **median** in unsorted datasets.
- Forms the basis of **introselect algorithm** (used in C++ STL's `nth_element`).
- Useful in **real-time systems** where predictable performance is crucial.
- Applied in **computational geometry** and **data analysis** tasks requiring robust selection.

Thank You