# Median Analysis Report

## Algorithm Interface

```java
public interface MedianAlgorithm {
  /**
   * @return the name of the algorithm
   */
  String getName();

  /**
   * Finds the median of the given array
   *
   * @param array the input array
   * @return the median value
   */
  int calculateMedian(int[] array);
}
```

## Naive Sorting Method

```java
public class NaiveMedian implements MedianAlgorithm {

  private String name = "Naive Median";

  @Override
  public String getName() {
    return name;
  }

  @Override
  public int calculateMedian(int[] array) {
    if (array == null || array.length == 0) {
      throw new IllegalArgumentException("Array cannot be null or empty");
    }
    int size = array.length;
    int medianIndex = (size - 1) / 2;
    java.util.Arrays.sort(array);
    return array[medianIndex];
  }
}
```

## Randomized Quick Select Method

```java
public class RandomizedMedian implements MedianAlgorithm {
  private String name = "Randomized Median";
  private Random random = new Random();
  @Override
  public String getName() {
    return name;
  }
  @Override
  public int calculateMedian(int[] array) {
    if (array == null || array.length == 0) {
      throw new IllegalArgumentException("Array cannot be null or empty");
    }
    int size = array.length;
    int medianIndex = (size - 1) / 2;
    return quickSelect(array, 0, size - 1, medianIndex);
  }

  private int quickSelect(int[] array, int low, int high, int k) {
    if (low == high) {
      return array[low];
    }
    int pivotIndex = partition(array, low, high);
    if (pivotIndex == k) {
      return array[k];
    } else if (pivotIndex < k) {
      return quickSelect(array, pivotIndex + 1, high, k);
    } else {
      return quickSelect(array, low, pivotIndex - 1, k);
    }
  }

  int partition(int[] array, int low, int high) {
    randomPivoting(array, low, high);
    int pivot = array[high];
    int i = low - 1;
    for (int j = low; j < high; j++) {
      if (array[j] <= pivot) {
        i++;
        swap(array, i, j);
      }
    }
    swap(array, i + 1, high);
    return i + 1;
  }

  void randomPivoting(int[] array, int low, int high) {
    int rand = low + random.nextInt(high - low + 1);
    swap(array, rand, high);
  }
}
```

# Median of medians Method

```java
public class DeterministicMedian implements MedianAlgorithm {
  private String name = "Deterministic Median";
  private int groupSize = 5;
  @Override
  public String getName() {
    return name;
  }
  @Override
  public int calculateMedian(int[] array) {
    if (array == null || array.length == 0) {
      throw new IllegalArgumentException("Array cannot be null or empty");
    }
    int size = array.length;
    int medianIndex = (size - 1) / 2;
    return medianOfMedians(array, 0, size - 1, medianIndex);
  }
  private int medianOfMedians(int[] array, int low, int high, int k) {
    int n = high - low + 1;
    if (n <= groupSize) {
      insertionSort(array, low, high);
      return array[low + k];
    }
    int numOfGroups = (n + groupSize - 1) / groupSize;
    int[] medians = new int[numOfGroups];
    for (int i = 0; i < numOfGroups; i++) {
      int currLow = low + i * groupSize;
      int currHigh = Math.min(currLow + groupSize - 1, high);
      insertionSort(array, currLow, currHigh);
      medians[i] = array[currLow + (currHigh - currLow) / 2];
    }
    int medianOfMedians = medianOfMedians(medians, 0, numOfGroups - 1, numOfGroups / 2);
    int pivotIndex = partition(array, low, high, medianOfMedians);
    if (pivotIndex - low == k) {
      return array[pivotIndex];
    } else if (pivotIndex - low > k) {
      return medianOfMedians(array, low, pivotIndex - 1, k);
    } else {
      return medianOfMedians(array, pivotIndex + 1, high, k - (pivotIndex - low + 1));
    }
  }

  private int partition(int[] array, int low, int high, int pivot) {
    int pivotIndex = findPivotIndex(array, low, high, pivot);
    swap(array, pivotIndex, high);

    int pivotValue = array[high];
    int i = low - 1;
```

```
    for (int j = low; j < high; j++) {
      if (array[j] <= pivotValue) {
        i++;
        swap(array, i, j);
      }
    }
    swap(array, i + 1, high);
    return i + 1;
  }

  private int findPivotIndex(int[] array, int low, int high, int pivot) {
    for (int i = low; i <= high; i++) {
      if (array[i] == pivot) {
        return i;
      }
    }
    return low;
  }
}
```

# Analysis Results

```
================================================================================
MEDIAN ALGORITHM PERFORMANCE COMPARISON
================================================================================
Data Size     Randomized Median     Deterministic Median     Naive Median
--------------------------------------------------------------------------------
10                    2.250µs                  2.910µs              50.708µs
100                  61.022µs                 21.225µs              46.633µs
1.0K                 20.749µs                 31.162µs              71.843µs
10.0K               156.268µs                824.643µs             436.391µs
100.0K              765.512µs                  1.830ms               3.694ms
1.0M                  5.439ms                 19.256ms              45.264ms
10.0M                58.078ms                196.311ms             545.865ms
```

**Randomized Median**

- **Expected Complexity: O(n) (average case)**
- **Comment: Fastest in practice — low overhead and performs well even on large datasets. Rarely hits bad cases due to random pivot selection.**

---

**Deterministic Median (Median of Medians)**

- **Expected Complexity: O(n) (worst case, guaranteed)**
- **Comment: Slower than randomized due to higher constant factors from recursive median calculations, but guarantees stable linear performance with no bad cases.**

---

**Naive Median (Sorting-based)**

- **Expected Complexity: O(n log n)**
- **Comment: Simplest but least efficient — sorts the entire array unnecessarily. Performance degrades quickly as input size grows.**