**Problem 1: Sorting Bot's QuickSort Challenge**

In a sprawling data center where thousands of IoT devices constantly stream in sensor data, a powerful AI bot named Sorto is tasked with organizing this data in real-time. The data includes temperature readings, pressure changes, light intensity values, and more — all of which must be quickly arranged to allow for instant analytics and automated triggers.

Sorto uses a divide-and-conquer strategy known as QuickSort, which allows it to sort large volumes of data in-place, minimizing memory usage. However, Sorto faces a unique constraint — due to real-time processing, it must complete sorting without creating any extra space or backup copies of data. Your mission is to implement Sorto’s in-place QuickSort routine that ensures real-time performance and efficient memory utilization. This function is part of an AI system that interfaces with many devices and must ensure data reliability under extreme processing speed. The real challenge lies not in the logic but in the execution — Sorto must perform consistently under fluctuating workloads and varying input patterns.

**Input Format:**

- First line: Integer `n` (number of sensor readings)

- Second line: `n` space-separated integers (sensor values)

**Output Format:**

- Sorted array in ascending order (space-separated)

**Sample Input:**

6

9 4 6 2 10 1

**Sample Output:**

1 2 4 6 9 10

**Sample Input (Already Sorted):**

5

1 2 3 4 5

**Sample Output:**

1 2 3 4 5

**Sample Input (Reverse Sorted):**

5

5 4 3 2 1

**Sample Output:**

1 2 3 4 5

**Problem 2: Playlist Organizer – MergeSort with Linked List**

A leading music streaming company tracks users' recently played songs through a linked list structure. Unlike arrays, these playlists can be dynamically extended, reordered, or filtered as the user listens. Each node in the list represents a unique track ID. As part of a feature upgrade, the development team wants to sort each user’s recent playlist based on play frequency, allowing the platform to generate smarter recommendations.

Internally, the recommendation engine interacts with this playlist in real-time, retrieving sorted tracks on-the-fly to influence the next song queue. However, converting this list into an array would introduce memory overhead and delay. This sorting must occur in-place on the linked list, preserving node connections and ensuring seamless updates without freezing the application. Your task is to sort this linked list using MergeSort, a stable algorithm particularly effective in scenarios with limited memory fragmentation.

**Input Format:**

- First line: Integer `n` (number of songs in the list)

- Second line: `n` space-separated integers (track IDs)

**Output Format:**

- Sorted linked list (space-separated values)

**Sample Input:**

5

9 1 4 7 2

**Sample Output:**

1 2 4 7 9

**Sample Input (Single Element):**`

1

42

**Sample Output:**

42

**Sample Input (Already Sorted):**

4

10 20 30 40

**Sample Output:**

10 20 30 40

**Problem 3: Statistics Engine – Median of Merged Sorted Arrays**

A national statistics bureau collects separate datasets from two different regions for health analysis. The data comes pre-sorted due to regional aggregation, and each dataset may have different sizes depending on the population density.

The analytics engine must find the \*\*median\*\* of the combined dataset to determine the central tendency of critical metrics like body temperature, oxygen levels, or heartbeat rates. However, due to memory limitations and strict regulatory constraints, you’re not allowed to merge the two arrays physically. Instead, your solution must find the median \*\*without fully merging the data\*\*, making the process computationally elegant and efficient.

This is vital during large-scale health crises when insights are needed instantly from streaming data — computing medians on-the-fly without extra storage is not just a performance optimization, it’s a necessity.

**Input Format:**

- First line: Integer `n` (size of first dataset)

- Second line: `n` space-separated integers (sorted)

- Third line: Integer `m` (size of second dataset)

- Fourth line: `m` space-separated integers (sorted)

**Output Format:**

- Median (float if needed)

**Sample Input:**

2

1 3

2

2 4

**Sample Output:**

2.5

**Sample Input (Even size):**

4

1 2 3 6

4

4 5 7 8

**Sample Output:**

4.5

**Sample Input (Odd size):**

3

1 3 5

2

2 4

**Sample Output:**

3

**Problem 4: Tournament Winner – K-th Smallest Using QuickSelect**

You’re building the ranking algorithm for a high-stakes global coding tournament. Participants are ranked based on their problem-solving scores, and the leaderboard needs to support real-time lookup of the k-th best performance.

To optimize speed and reduce unnecessary sorting, you decide to implement the \*\*QuickSelect\*\* algorithm, which efficiently finds the K-th smallest number without sorting the entire dataset. This enables near-instant ranking retrieval during live competitions, even as new scores continue to pour in.

Behind the scenes, this method is used not only for ranking participants but also for threshold calculations like identifying top-tier performers for prize distributions. The algorithm must handle large bursts of incoming scores and respond in milliseconds. A wrong implementation could lead to unjust results, skewing the fairness of the entire competition.

**Input Format:**

- First line: Integer `n` (number of participants)

- Second line: `n` space-separated scores

- Third line: Integer `k`

**Output Format:**

- The K-th smallest score

**Sample Input:**

6

7 10 4 3 20 15

3

**Sample Output:**

7

**Sample Input (K=1):**

5

8 5 3 2 10

1

**Sample Output:**

2

**Sample Input (K=n):**

4

10 20 30 40

4

**Sample Output:**

40

**Problem 5: Spy Decoder – Search in Rotated Sorted Array with Duplicates**

A covert agency stores encrypted messages in rotated sorted arrays, making it difficult for intruders to decode without the proper algorithm. Each message ID is stored in a sorted array that has been rotated at an unknown pivot to enhance data obfuscation. Additionally, duplicate entries are allowed to mislead attackers trying to detect patterns.

An intelligence analyst is tasked with determining whether a given message ID exists within one of these rotated datasets. However, their access is limited to logarithmic time complexity due to computational security layers in place. Linear scanning is considered insecure and traceable.

This analyst operates in highly constrained environments — sometimes from field laptops with limited memory and processing power. Their tools must provide answers with speed and accuracy even in the presence of large, tampered datasets.

**Input Format:**

- First line: Integer `n`

- Second line: `n` space-separated integers (rotated sorted with duplicates)

- Third line: Integer `x` (message ID to search)

**Output Format:**

- `true` if found, otherwise `false`

**Sample Input:**`

7

2 5 6 0 0 1 2

0

**Sample Output:**

true

**Sample Input (Not Found):**

6

1 1 3 1 1 1

2

**Sample Output:**

false

**Sample Input (Found with Duplicates):**

6

1 1 3 1 1 1

3

**Sample Output:**

true