🔍 Finding the Kth Smallest Element using a Max Heap (Priority Queue)

✅ Problem Statement:

Find the **kth smallest element** from an unsorted array in **efficient time complexity**, using a **heap-based approach**.

🧠 Core Idea (Logic):

We use a **Max Heap** (implemented in C++ using `priority_queue<int>`) to maintain the **k smallest elements** at any given point.

- Push elements into the Max Heap one by one.
- If the heap size exceeds `k`, pop the top element (which is the largest among the current elements in the heap).
- After processing all elements, the heap will contain exactly `k` smallest elements.
- The **top of the Max Heap** will be the **kth smallest element**, because the heap contains the k smallest values, and the largest among them is at the top.

🧪 Example Input:

```
vector<int> v = {10, 20, -4, 6, 1, 24, 105, 118};
int k = 4;  // Find the 4th smallest element
```

🧾 C++ Code:

```
#include<iostream>
#include<vector>
#include<queue>
using namespace std;

int main() {
  vector<int> v = {10, 20, -4, 6, 1, 24, 105, 118};
  int k = 4;  // Find 4th smallest element

  // Max Heap to store k smallest elements
  priority_queue<int> pq;

  for(int i = 0; i < v.size(); i++) {
    pq.push(v[i]);          // Step 1: Push element
    if(pq.size() > k)       // Step 2: If size exceeds k
      pq.pop();             // Remove the largest (top)
  }

  cout << pq.top();         // Top element is the kth smallest
  return 0;
}
```

📊 Dry Run:

Let's take: `v = {10, 20, -4, 6, 1, 24, 105, 118}`, and `k = 4`

- Push 10 → [10]
- Push 20 → [20, 10]
- Push -4 → [20, 10, -4]
- Push 6 → [20, 10, -4, 6]
- Push 1 → [20, 10, -4, 6, 1] → Size > 4 → pop → [10, 6, -4, 1]
- Push 24 → [24, 10, -4, 1, 6] → pop → [10, 6, -4, 1]

- Push 105 → [105, 10, -4, 1, 6] → pop → [10, 6, -4, 1]
- Push 118 → [118, 10, -4, 1, 6] → pop → [10, 6, -4, 1]

✅ At end, PQ contains 4 elements: [10, 6, -4, 1]

✅ Top (10) is the **4th smallest element**

🕒 Time & Space Complexity:

- **Time Complexity:**
  - Each `push` and `pop` operation on the heap: **O(log k)**
  - Loop runs for **n** elements → **O(n log k)**
- **Space Complexity:**
  - Max heap holds **k elements** → **O(k)**

🔁 Use Case:

- Efficient when **k is small** compared to **n**
- Better than sorting the full array (**O(n log n)**)

🔚 Final Note:

This is a **clever trick** to use the **Max Heap** to track smallest elements.

By always ejecting the **largest among smallest**, you make sure that at the end, the top of the heap is exactly the **kth smallest element**.
