Top 'K' Elements Pattern

This pattern is used to efficiently find the **largest**, **smallest**, or **most frequent** K elements in a collection (array, stream, etc.)—not by sorting the entire array, but by using a **heap** (priority queue) for quick access to min/max elements.

- **Min-Heap**: Used to keep track of the **K largest** elements.
- **Max-Heap**: Used to keep track of the **K smallest** or **K most frequent** elements.

Top 'K' Elements Pattern

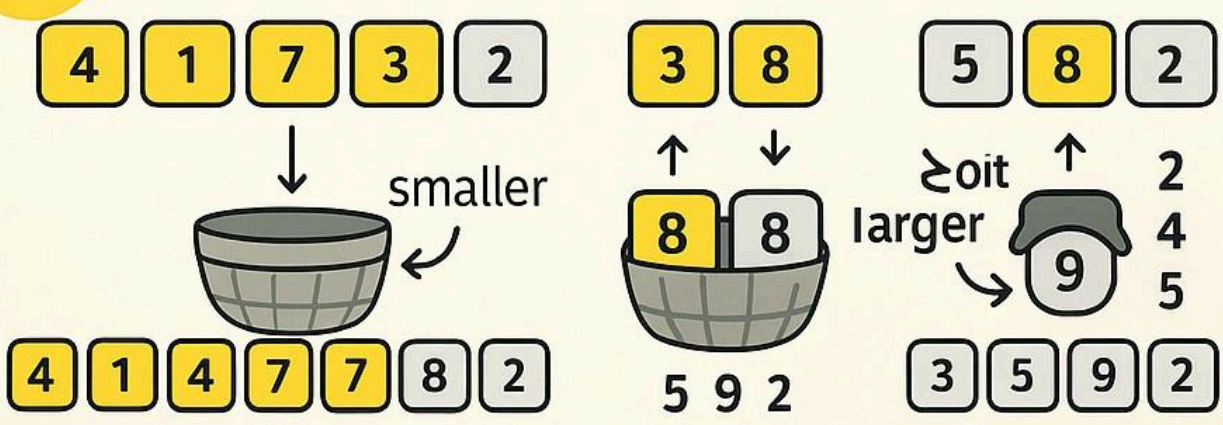
K Largest Elements Using Min-Heap

Input Array: [4, 1, 7, 3, 8, 9, 2]

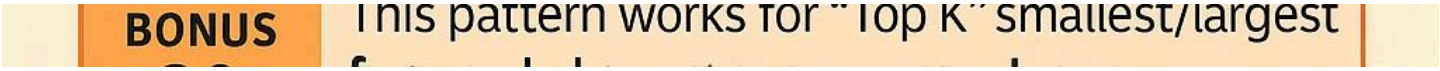
Task: Find the 3 largest elements ithe array.

STEP 1 Use a **min-heap of size K** to keep track of the largest K elements.

STEP 2 Process each element in the array



AT HE 3 At the end, the heap contains the K largest elements (order may vary)



Classic Problem: Find the K Largest Elements in an Array

Problem Statement:
Given an array and an integer K, find the K largest elements in the array.

Explanation

1. **Use a Min-Heap of size K:**
   - Heap holds the current K largest elements.
   - The smallest among them is always at the top.
2. **For each element in the array:**
   - If the heap has less than K elements, insert it.
   - Otherwise, if the current element is larger than the smallest in the heap, remove the smallest and insert the new element.
3. **At the end, the heap contains the K largest elements.**

C Code Example: K Largest Elements Using Min-Heap

C does not have a built-in heap/priority queue, but we can use a simple array to manually implement a fixed-size min-heap for small K. Here's a simple code for K=3 for clarity.

```
#include <stdio.h>
#include <stdlib.h>

// Swap helper
void swap(int* a, int* b) {
    int t = *a; *a = *b; *b = t;
}

// Heapify up for min-heap
void heapifyUp(int heap[], int idx) {
    while (idx > 0) {
        int parent = (idx - 1) / 2;
        if (heap[parent] > heap[idx]) {
            swap(&heap[parent], &heap[idx]);
            idx = parent;
        } else break;
    }
}

// Heapify down for min-heap
void heapifyDown(int heap[], int size, int idx) {
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;
    int smallest = idx;

    if (left < size && heap[left] < heap[smallest]) smallest = left;
    if (right < size && heap[right] < heap[smallest]) smallest = right;

    if (smallest != idx) {
        swap(&heap[smallest], &heap[idx]);
```

```
        heapifyDown(heap, size, smallest);
    }
}

void kLargestElements(int arr[], int n, int k) {
    int heap[k];
    int heapSize = 0;

    for (int i = 0; i < n; i++) {
        if (heapSize < k) {
            heap[heapSize++] = arr[i];
            heapifyUp(heap, heapSize - 1);
        } else if (arr[i] > heap[0]) {
            heap[0] = arr[i];
            heapifyDown(heap, k, 0);
        }
    }

    // Print K largest elements (in heap, not sorted)
    printf("Top %d largest elements: ", k);
    for (int i = 0; i < k; i++) {
        printf("%d ", heap[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {4, 1, 7, 3, 8, 5, 9, 2};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k = 3;

    kLargestElements(arr, n, k);

    return 0;
}
```

How does this work?

- The heap always keeps the smallest of the current K largest at the top.
- If you find a bigger number, you pop the smallest and insert the new one.
- At the end, the heap holds the K largest.

Where is this pattern useful?

- Finding top K elements (largest, smallest, or most frequent)
- K closest points to origin (geometry problems)
- Streaming data where you want a running top K

Practice Challenge

- Try finding the **K smallest elements** using a **max-heap**.
- Try finding the **K most frequent numbers** (you'll need a frequency table + heap).