Topic Introduction

Today, we're diving into a classic interview theme: **Top K Elements** — problems where you need to efficiently find the top K items according to some criteria (largest, most frequent, or closest). This concept is a cornerstone in coding interviews because it tests your mastery of heaps (priority queues), quickselect, and sometimes binary search or sliding windows.

What is the "Top K" Pattern?

The "Top K" pattern is about retrieving the K highest (or lowest) items from a collection, without fully sorting the entire dataset. The most common tools for this are **heaps** (often min-heaps or max-heaps), and occasionally quickselect (a selection algorithm related to quicksort). Both allow you to avoid the O(n log n) cost of a full sort, achieving O(n log k) or O(n) in some cases.

How does it work?

- **Heap Approach:**

Maintain a heap (size K) that tracks the top items as you scan the list. For top K largest, use a min-heap: if you see a bigger item, pop the smallest and push the new one. For top K smallest, use a max-heap.

- **Quickselect:**

Partition the array so the Kth largest (or smallest) is in its sorted position. This is average O(n) time, but trickier to implement.

When is this useful in interviews?

When you see "find the K largest/smallest/frequent/closest" in a problem, immediately consider heaps or quickselect. Interviewers love these because they test your data structure knowledge, algorithmic intuition, and efficiency.

Quick Example:

Given `[7, 10, 4, 3, 20, 15]` and K = 3, what are the three largest elements?

- Full sort: `[3, 4, 7, 10, 15, 20]` → last three: `[10, 15, 20]`
- Min-heap: Build a heap of size 3 as you go, always keeping the three largest seen so far.

Why These 3 Problems?

- **Top K Frequent Elements** (frequency + heap)
- **Kth Largest Element in an Array** (quickselect or heap)
- **Find K Closest Elements** (binary search, heap, or two-pointer window)

All three are classic "Top K" problems, but each adds its own twist: counting, selection, or proximity. You'll see how heaps, quickselect, and binary search naturally fit different flavors of this pattern.

Problem 1: Top K Frequent Elements

Leetcode 347: Top K Frequent Elements

Problem Statement:

Given an array of integers `nums` and an integer `k`, return the `k` most frequent elements in any order.

Example:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`

Output: `[1,2]`

Explanation: `1` appears 3 times, `2` appears 2 times, `3` appears once. The top 2 are `1` and `2`.

Let's Try Another Example:

Input: `nums = [4,4,4,6,6,1,1,1,3]`, `k = 2`

What should the output be? (Pen and paper time!)

Brute-force Approach:

Count the frequency of each number, then sort all numbers by their frequency and pick the top k.

- Counting: O(n)
- Sorting: O(n log n)
- Total: O(n log n)

Optimal Approach: Heap

We'll use a hashmap to count frequencies, and a min-heap of size `k` to track the k most frequent.

Step-by-Step Logic:

- Count frequencies with a hashmap.
- Push each (frequency, number) pair onto a min-heap.

If the heap grows bigger than k, pop the smallest frequency.

- At the end, the heap contains the k most frequent numbers.

Python Solution:

```
import heapq
from collections import Counter

def topKFrequent(nums, k):
    # Step 1: Count frequencies
    freq = Counter(nums)  # e.g. {1:3, 2:2, 3:1}

    # Step 2: Use a min-heap of size k
    # Heap elements: (frequency, number)
    heap = []
    for num, count in freq.items():
        heapq.heappush(heap, (count, num))
        if len(heap) > k:
            heapq.heappop(heap)

    # Step 3: Extract numbers from heap
    result = [num for count, num in heap]
    return result
```

Time Complexity:

- Counting: O(n)
- Heap operations: O(n log k)

- Total: O(n log k)

Space Complexity: O(n) for the hashmap and the heap (up to k elements).

Code Breakdown:

- We use `Counter` to tally frequencies.
- For each unique number, we push its (frequency, number) onto the heap.
- If the heap exceeds k, we remove the smallest frequency.
- At the end, the heap holds the k most frequent numbers.

Trace Example:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`

- Frequencies: `{1:3, 2:2, 3:1}`
- Heap after processing: `[(2, 2), (3, 1)]` (holds the two most frequent)
- Output: `[2, 1]` (order may vary)

Try This Test Case:

Input: `nums = [5,3,1,1,1,3,73,1]`, `k = 2`

What should the output be?

Take a moment to solve this on your own before jumping into the solution.

Alternative:

There's also a bucket sort O(n) solution, but the heap method is most common and useful for larger k.

Reflect:

Did you know this could also be solved using a bucket sort approach? Try implementing that version once you're comfortable with heaps!

Problem 2: Kth Largest Element in an Array

Leetcode 215: Kth Largest Element in an Array

Problem Statement:

Given an unsorted array `nums`, find the Kth largest element. (Note: The Kth largest is not necessarily distinct.)

Example:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`

Output: `5`

Explanation: 6 is largest, 5 is second largest.

Why is this similar?

Again, we want a "top K" item, but now just the single Kth largest element, not all top K.

Brute-force Approach:

Sort the array (O(n log n)), then pick the Kth largest: `nums[-k]`.

Optimal Approach: Min-Heap or Quickselect

- **Min-Heap approach:** Maintain a min-heap of size k. As you scan, keep only the k largest seen so far. The root is the Kth largest.

- **Quickselect:** Partition the array so the Kth largest lands in its place. Average O(n) time.

Let's focus on the Min-Heap method here:

Step-by-step:

- Initialize a min-heap.
- For each number, push it onto the heap.
- If the heap exceeds size k, pop the smallest.
- After all elements, the heap root is your answer.

Example Dry Run:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`

- Heap after: [5, 6]
- Kth largest: 5

Another Test Case:

Input: `nums = [7,10,4,3,20,15]`, `k = 3`

What is the output?

Pseudocode:

```
function findKthLargest(nums, k):
    minHeap = new empty heap
    for num in nums:
        push num onto minHeap
        if size of minHeap > k:
            pop minHeap
    return minHeap[0]
```

Trace Example:

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`

- Heap after: [4, 5, 5, 6]
- Output: 4

Time Complexity:

- O(n log k)

Space Complexity:

- O(k)

Try This Test Case:

Input: `nums = [10, 9, 8, 7, 6, 5]`, `k = 5`

What is the output?

Problem 3: Find K Closest Elements

Leetcode 658: Find K Closest Elements

Problem Statement:

Given a sorted array `arr`, a number `x`, and an integer `k`, return the k elements closest to x (sorted in ascending order). If two numbers are equally close, prefer the smaller one.

What's new here?

Instead of simply finding the largest or most frequent, we now care about proximity to a target value. The input is sorted, making binary search and windowing possible.

Brute-force Approach:

For each element, compute its distance to x. Sort by distance, and return the top k.

- O(n log n) time.

Optimal Approach: Binary Search + Sliding Window

Because the array is sorted, we can find the optimal window of size k using binary search.

Step-by-Step:

- Set `left = 0`, `right = len(arr) - k`.
- While `left < right`:
  - Let `mid = (left + right) // 2`.
  - If `x - arr[mid] > arr[mid + k] - x`, move `left = mid + 1`.
  - Else, move `right = mid`.
- Return `arr[left:left + k]`.

Why does this work?

We're searching for the leftmost starting index of a window of size k, whose elements are closest to x.

Pseudocode:

```
function findClosestElements(arr, k, x):
    left = 0
    right = length(arr) - k
    while left < right:
        mid = (left + right) // 2
        if x - arr[mid] > arr[mid + k] - x:
            left = mid + 1
        else:
            right = mid
    return arr[left:left + k]
```

Example:

Input: `arr = [1,2,3,4,5]`, `k = 4`, `x = 3`

- Output: `[1,2,3,4]`

Trace Example:

- `left = 0`, `right = 1`
- Check mid = 0: 3 - 1 = 2, 5 - 3 = 2, so right = 0.
- Output: arr[0:4] = [1,2,3,4]

Try This Test Case:

Input: `arr = [1,2,3,4,5,6,7]`, `k = 3`, `x = 5`

What is the output?

Complexity:

- Time: O(log(n - k) + k)
- Space: O(1) extra (ignoring output)

Alternative:

You can also use a max-heap to keep the k closest, but binary search is optimal for sorted arrays.

Nudge:

Can you think of cases where a heap might be better than binary search for this problem? Try implementing both!

Summary and Next Steps

These three problems are all classic "Top K" challenges, but each highlights a different flavor:

- Counting and frequency (Top K Frequent Elements)
- Selection and ordering (Kth Largest Element)
- Closeness and proximity (Find K Closest Elements)

Key Patterns:

- Use heaps (priority queues) to efficiently track top K items.
- Use quickselect for the Kth largest/smallest in O(n) time.
- For sorted arrays, binary search + window is powerful for proximity-based problems.

Common Pitfalls:

- Mixing up min-heap and max-heap: For largest elements, use a min-heap of size K!
- Not leveraging sorted input: When arrays are sorted, binary search is often better than heaps.
- Forgetting to handle ties or output order as specified.

Action List

- Try solving all three problems on your own, including the code-provided one.
- For Problem 2 and 3, try an alternative approach (quickselect for Kth largest, or heap for closest elements).
- Look for other "Top K" problems — practice will help you internalize these patterns.
- When reviewing your solutions, compare for performance and edge cases.
- If you get stuck, break down the problem, draw diagrams, and revisit the pattern explanations.

Keep practicing — the more you see these patterns, the faster you'll spot them in interviews!