

Top K Frequent Elements

HashMap + Min-Heap combo — a pattern you'll use everywhere ■

● MEDIUM

■ Hook

Which words appear most often in a million-line log file? Which products are bought most? Which IP addresses hit your server most? **Top-K Frequency** is everywhere. This problem teaches the canonical two-step pattern: **count with HashMap, rank with Min-Heap**.

■ Core Intuition — Count Then Rank

Step 1: Count frequencies using a HashMap. Step 2: Use a min-heap of size K keyed by frequency. Identical to Kth Largest — but now the 'value' we compare is frequency, not the number itself.

■ Alternative: Bucket Sort $O(n)$

Since frequencies range from 1 to n, you can bucket-sort: create an array of lists indexed by frequency. Fill from end until you have K elements. $O(n)$ time and space — optimal for this problem.

■ Visual — [1,1,1,2,2,3], K=2

```
STEP 1 — Build frequency map:
[1,1,1,2,2,3] → {1:3, 2:2, 3:1}
STEP 2 — Min-heap (keyed by frequency), size K=2:
Add (1,3): heap=[(1,3)]           size=1
Add (2,2): heap=[(2,2),(1,3)]      size=2 [ordered by freq]
Add (3,1): freq=1 < heap.peek().freq=2 → poll (2,2)? NO!
           Wait — 1 < 2 so (3,1) is weaker. Poll? Let's check:
           heap size=2 already. Add (3,1), size=3 > K:
           poll() removes (3,1) since freq=1 is minimum.
           heap=[(2,2),(1,3)]
FINAL heap contains: {1 (freq=3), 2 (freq=2)}
Extract all: [1, 2]
★ ANSWER = [1, 2]
BUCKET SORT VISUAL:
freq: [_, [3], [2], [1]]
idx:  0   1   2   3
Read from right: 1(freq3), 2(freq2) → K=2 done. [1,2] ✓
```

■ Code Implementation (Java) — Both Approaches

```

import java.util.*;
public class TopKFrequent {
    // Approach 1: HashMap + Min-Heap O(n log K)
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer,Integer> freq = new HashMap<>();
        for (int n : nums) freq.merge(n, 1, Integer::sum);
        // Min-heap ordered by frequency
        PriorityQueue<int[]> minHeap =
            new PriorityQueue<>((a, b) -> a[1] - b[1]);
        for (Map.Entry<Integer,Integer> e : freq.entrySet()) {
            minHeap.offer(new int[]{e.getKey(), e.getValue()});
            if (minHeap.size() > k) minHeap.poll();
        }
        int[] result = new int[k];
        for (int i = k-1; i >= 0; i--) result[i] = minHeap.poll()[0];
        return result;
    }

    // Approach 2: Bucket Sort O(n)
    public int[] topKFrequentBucket(int[] nums, int k) {
        Map<Integer,Integer> freq = new HashMap<>();
        for (int n : nums) freq.merge(n, 1, Integer::sum);
        List<Integer>[] bucket = new List[nums.length + 1];
        for (int key : freq.keySet()) {
            int f = freq.get(key);
            if (bucket[f] == null) bucket[f] = new ArrayList<>();
            bucket[f].add(key);
        }
        List<Integer> result = new ArrayList<>();
        for (int i = bucket.length-1; i >= 0 && result.size() < k; i--)
            if (bucket[i] != null) result.addAll(bucket[i]);
        return result.stream().mapToInt(Integer::intValue).toArray();
    }
}

```

■ ■ Time & Space Complexity

■ TIME COMPLEXITY

$O(n \log K)$ / $O(n)$

■ SPACE COMPLEXITY

$O(n)$

HashMap: $O(n)$. Heap approach: $O(n \log K)$ time, $O(n+K)$ space. Bucket sort: $O(n)$ time, $O(n)$ space — optimal when $K \approx n$.

■ ■ Common Mistakes

■ ■ Custom comparator direction

The heap comparator $(a, b) \rightarrow a[1] - b[1]$ orders by ASCENDING frequency (min-heap by freq). Getting the direction wrong gives you the bottom-K instead of top-K. Verify your comparator!

■ Practice Problems

#	Problem	Diff	Pattern
1	Top K Frequent Elements (LC 347)	Medium	HashMap + Min-Heap
2	Top K Frequent Words (LC 692)	Medium	HashMap + Min-Heap + lexicographic
3	Sort Characters By Frequency (LC 451)	Medium	HashMap + Max-Heap
4	K Closest Points to Origin (LC 973)	Medium	Max-Heap by distance
5	Find K Pairs with Smallest Sums (LC 373)	Medium	Min-Heap

■ Quick Revision

- ✓ Step 1: Build frequency map with HashMap
- ✓ Step 2: Min-heap of size K keyed by frequency
- ✓ Custom comparator: $(a,b) \rightarrow a[1] - b[1]$ for freq ordering
- ✓ Bucket sort alternative: $O(n)$ time/space
- ✓ Poll K times from max-heap OR collect from min-heap of size K
- ✓ Pattern reuses everywhere: top-K by any metric

■ MENTAL MODEL — Remember This Forever

Two-phase pattern: CENSUS then ELECTION. Phase 1 (Census): Count how many votes each candidate (element) got — that's your HashMap. Phase 2 (Election): Run a small election among K seats. The min-heap acts as the K elected seats — a new candidate with more votes can unseat the current weakest seat-holder.