Find Top K Elements with Heap — Coding Interview Notes (Light Theme)

General Pattern Template

```
import heapq

def fn(arr, k):
    heap = []
    for num in arr:
        # do some logic to push onto heap according to problem's criteria
        heapq.heappush(heap, (CRITERIA, num))
        if len(heap) > k:
            heapq.heappop(heap)

    return [num for num in heap]
```

Concept:
To extract the **top K** items by a criterion (largest values, most frequent, closest to target, etc.), maintain a **size-k heap** while streaming through the data. In Python, heapq is a *min-heap* by default.

Recipe: Push tuples of the form (key, item), where key encodes the ranking criterion. Keep heap size ≤ k; pop when size exceeds k so the heap stores the current top K by your criterion.

Complexity: O(n log k) time, O(k) space.

Key Ideas

1. Python's heapq is a min-heap; store keys so the smallest key corresponds to the worst among current top K.
2. For 'largest K', use key = value (min-heap keeps the smallest of the top K at the root).
3. For 'smallest K', you can store negative values or use a max-heap emulation with (-key, item).
4. For frequency problems, precompute counts and heapify pairs (count, item).
5. At the end, the heap contains K items; sort if you need ordered output.

Example 1: Top K Largest Elements

Goal: Return the K largest numbers from the array.
Approach: Maintain a size-k *min-heap* of values. Pop when size exceeds k.

```
import heapq

def top_k_largest(nums, k):
    heap = []
```

```
    for x in nums:
        heapq.heappush(heap, x)      # key is the value itself
        if len(heap) > k:
            heapq.heappop(heap)       # remove current smallest among top-K
    # heap now holds k largest in arbitrary heap order
    return sorted(heap, reverse=True)

# Example
print(top_k_largest([3,1,5,12,2,11], 3))  # Output: [12, 11, 5]
```

Example 2: Top K Frequent Elements

Goal: Return the K elements with the highest frequency.
Approach: Count with a dictionary, then keep a size-k min-heap of (count, value).

```
import heapq
from collections import Counter

def top_k_frequent(nums, k):
    freq = Counter(nums)
    heap = []
    for val, cnt in freq.items():
        heapq.heappush(heap, (cnt, val))  # min-heap by count
        if len(heap) > k:
            heapq.heappop(heap)
    # Extract values and sort by count descending (optional)
    return [val for cnt, val in sorted(heap, key=lambda x: -x[0])]

# Example
print(top_k_frequent([1,1,1,2,2,3], 2))  # Output: [1, 2]
```

Example 3: K Closest Numbers to Target

Goal: Return K numbers closest to a target *t* by absolute difference.
Approach: Keep a size-k *max-heap* of (distance, value) by pushing negative distances.

```
import heapq

def k_closest(nums, k, t):
    heap = []  # will store (-distance, value) so root is the farthest among kept
    for x in nums:
        key = abs(x - t)
        heapq.heappush(heap, (-key, x))
        if len(heap) > k:
            heapq.heappop(heap)  # remove farthest among the kept
    # Extract values; result order not guaranteed unless sorted
    return [x for _, x in sorted(heap, key=lambda p: abs(p[1] - t))]

# Example
print(k_closest([5,6,7,8,9], 3, 7))  # Output: [7, 6, 8] (order may vary)
```

Summary Table

ProblemHeap ContentKey/TrickComplexity K largestMin-heap of valuesPop when size > kO(n log k) K most frequentMin-heap of (count, val)Use Counter, key by countO(u log k) (u=#unique) K closest to tMax-heap via negativesPush (-|x-t|, x)O(n log k)