# 347. Top K Frequent Elements

**Divide and Conquer** Heap (Priority Queue) Hash Table **Bucket Sort** Medium Counting Quickselect ) Sorting <u>Array</u>

**Problem Description** 

## The LeetCode problem provides us with an integer array nums and an integer k. Our task is to find the k most frequent elements in

Intuition

the array. The "frequency" of an element is the number of times it occurs in the array. The problem specifies that we can return

To solve this problem, we need to count the occurrences of each element and then find the k elements with the highest frequencies. The natural approach is to use a hash map (or dictionary in Python) to achieve the frequency count efficiently. Once we have the frequency of each element, we want to retrieve the k elements with the highest frequency. A common data

structure to maintain the k largest or smallest elements is a heap. In Python, we use a min-heap by default, which ensures that

the smallest element is always at the top. The intuition behind the solution is: 1. Count the frequency of each element using a hash map.

- 3. If the heap exceeds size k, we remove the smallest item, which is automatically done because of the heap's properties. This ensures we only
- keep the k most frequent elements in the heap.

frequency of the number num in the array.

solution is O(n) to store the frequency map and the heap.

find the 2 most frequent elements in nums.

frequent elements in the heap.

entire frequency map, which could be much larger than k.

4. After processing all elements, we're left with a heap containing k elements with the highest frequency.

2. Iterate over the frequency map, adding each element along with its frequency as a tuple to a min-heap.

the result in any order, which means the sequence of the results does not matter.

- 5. We convert this heap into a list containing just the elements (not the frequencies) to return as our final answer. This approach is highly efficient as it allows us to keep only the k most frequent elements at all times without having to sort the
- Solution Approach

The implementation of the solution uses Python's Counter class from the collections module to calculate the frequency of each element in the nums array. Counter is essentially a hash map or a dictionary that maps each element to its frequency.

## Here's a step-by-step walkthrough of the implementation:

for v in hp].

First, we use Counter(nums) to create a frequency map that holds the count of each number in the nums array. Next, we initialize an empty min-heap hp as a list to store tuples of the form (frequency, num), where frequency is the

We iterate over each item in the frequency map and add a tuple (freq, num) to the heap using the heappush function.

- While we add elements to the heap, we maintain the size of the heap to not exceed k. If adding an element causes the heap size to become greater than k, we pop the smallest item from the heap using heappop. This is done to keep only the k most
- After we finish processing all elements, the heap contains k tuples representing the k most frequent elements. The least frequent element is on the top of the min-heap, while the k-th most frequent element is the last one in the heap's binary tree representation.
- The Counter efficiently calculates the frequencies of each element in O(n) time complexity, where n is the length of the input array. The heap operations (insertion and removal) work in O(log k) time, and since we perform these operations at most n times,

the total time complexity of the heap operations is  $0(n \log k)$ . Thus, the overall time complexity of the solution is  $0(n \log k)$ ,

with O(n) coming from the frequency map creation and O(n log k) from the heap operations. The space complexity of the

Finally, we build the result list by extracting the num from each tuple (freq, num) in the heap using a list comprehension: [v[1]

avoids having to sort large sets of data. **Example Walkthrough** 

Let's use a small example to illustrate the solution approach. Consider the array nums = [1,2,3,2,1,2] and k = 2. Our goal is to

This efficient implementation ensures we're not doing unnecessary work by keeping only the top k frequencies in the heap, and it

We first use Counter(nums) to create a frequency map. This gives us {1: 2, 2: 3, 3: 1} where the key is the number from nums and the value is its frequency.

We iterate over the frequency map and add each num and its frequency to hp. For example, (2, 1) for the number 1 with a

The heap should not exceed the size k. In our case, k is 2, which means after we add the third element (3, 2), we need to

We initialize an empty min-heap hp. It's going to store tuples like (frequency, num).

ensuring that less frequent elements have been popped off when the size limit was exceeded.

### frequency of 2. We use heappush to add the tuples to hp, so after this step hp might have [(1, 3), (2, 1)].

Solution Implementation

def topKFrequent(self, nums, k):

num\_frequencies = Counter(nums)

if len(min\_heap) > k:

heappop(min\_heap)

heappush(min\_heap, (freq, num))

# Count the frequency of each number in nums using Counter.

# Python's heapq module creates a min-heap by default.

# If the heap size exceeds k, remove the smallest frequency element.

# Initialize a min heap to keep track of top k elements.

**Python** 

class Solution:

pop the smallest frequency. So we end up with hp as [(2, 1), (3, 2)] after all the operations since (1, 3) would be the popped element because it had the lowest frequency.

The heap now contains the tuples for the 2 most frequent elements. The tuple with the smallest frequency is at the top,

Finally, to build our result list, we extract the number from each tuple in the heap. Using list comprehension [v[1] for v in

- hp] we get [1, 2], which are the elements with the highest frequency. This is our final result and we can return it. Following this approach, we implemented an efficient solution to the problem that avoids sorting the entire frequency map directly and instead maintains a heap of size k to track the k most frequent elements.
- from collections import Counter from heapq import heappush, heappop

 $min_heap = []$ # Iterate over the number-frequency pairs. for num, freq in num\_frequencies.items(): # Push a tuple of (frequency, number) onto the heap.

```
# Extract the top k frequent numbers by taking the second element of each tuple.
top_k_frequent = [pair[1] for pair in min_heap]
return top_k_frequent
```

Java

import java.util.\*;

import java.util.function.Function;

import java.util.stream.Collectors;

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
       // Create a Map to store the frequency of each number
       Map<Integer, Long> frequencyMap = Arrays.stream(nums)
                                                boxed() // box the ints to Integers
                                                .collect(Collectors.groupingBy(Function.identity(), // group by the number itsel;
                                                    Collectors.counting())); // count the frequency
       // Initialize a min-heap based on the frequency values
       Queue<Map.Entry<Integer, Long>> minHeap = new PriorityQueue<>(Comparator.comparingLong(Map.Entry::getValue));
       // Iterate over the frequency map
        for (Map.Entry<Integer, Long> entry : frequencyMap.entrySet()) {
           // Insert the current entry into the min-heap
           minHeap.offer(entry);
            // If the heap size exceeds 'k', remove the smallest frequency element
            if (minHeap.size() > k) {
                minHeap.poll();
       // Extract the top 'k' frequent numbers from the min-heap into an array
        return minHeap.stream()
                     .mapToInt(Map.Entry::getKey)
                      .toArray();
C++
#include <vector>
#include <queue>
#include <unordered_map>
using namespace std;
// Definition for a pair of integers
using IntPair = pair<int, int>;
class Solution {
public:
    // Function that returns the k most frequent elements from 'nums'.
```

```
// Return the top k frequent elements in ascending frequency order.
       return topKFrequentElements;
};
TypeScript
```

for (const num of nums) {

minHeap.pop();

vector<int> topKFrequent(vector<int>& nums, int k) {

unordered\_map<int, int> frequencyMap;

for (int value : nums) {

frequencyMap[value]++;

// Iterate over the frequency map.

int number = element.first;

if (minHeap.size() > k) {

minHeap.pop();

for (int i = 0; i < k; ++i) {

int frequency = element.second;

for (const auto& element : frequencyMap) {

minHeap.push({frequency, number});

// Prepare a vector to store the result.

function topKFrequent(nums: number[], k: number): number[] {

// Convert the Map into an array of key-value pairs

// Sort the array based on frequency in descending order

// Initialize an array to hold the top k frequent elements

# Count the frequency of each number in nums using Counter.

# Push a tuple of (frequency, number) onto the heap.

# Python's heapq module creates a min-heap by default.

# If the heap size exceeds k, remove the smallest frequency element.

# Extract the top k frequent numbers by taking the second element of each tuple.

# Initialize a min heap to keep track of top k elements.

// Iterate k times and push the most frequent elements onto the topKElements array

const frequencyArray = Array.from(frequencyMap);

topKElements.push(frequencyArray[i][0]);

frequencyArray.sort((a, b) => b[1] - a[1]);

const topKElements: number[] = [];

for (let i = 0; i < k; i++) {

const frequencyMap = new Map<number, number>();

// Initialize a Map to hold the frequency of each number

// Iterate over the array of numbers and populate the frequencyMap

frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);

vector<int> topKFrequentElements(k);

// HashMap to store the frequency of each number in 'nums'.

// Increment the frequency count for each number in 'nums'.

// Min-heap to keep track of the top k frequent numbers.

// Retrieve the most frequent elements from the min-heap.

priority\_queue<IntPair, vector<IntPair>, greater<IntPair>> minHeap;

// Push the current number and its frequency to the min-heap.

// If the heap size exceeds k, remove the least frequent element.

// It stores pairs of (frequency, number) and orders by smallest frequency first.

// This ensures that the heap always contains the top k frequent elements.

topKFrequentElements[i] = minHeap.top().second; // Store the number, not the frequency.

```
// Return the top k frequent elements
return topKElements;
```

class Solution:

from collections import Counter

min\_heap = []

from heapq import heappush, heappop

def topKFrequent(self, nums, k):

num\_frequencies = Counter(nums)

if len(min heap) > k:

return top\_k\_frequent

Time and Space Complexity

heappop(min\_heap)

# Iterate over the number-frequency pairs.

for num, freq in num\_frequencies.items():

top\_k\_frequent = [pair[1] for pair in min\_heap]

heappush(min\_heap, (freq, num))

- **Time Complexity** The time complexity of the function is determined by several factors:
  - Heap Operations: The function then involves a for loop, iterating over the frequency counter's items, and performing heap operations. For each unique element (up to N unique elements), a heap push is performed, which has a time complexity of
- O(log K), as the size of the heap is maintained at k. In the worst case, there are N heap push and pop operations, each taking  $0(\log K)$  time. Therefore, the complexity due to heap operations is  $0(N * \log K)$ . The resulting overall time complexity is  $0(N + N * \log K)$ . However, since  $N * \log K$  is the dominant term, we consider the overall time complexity to be O(N \* log K).

**Counting Elements:** The function begins with cnt = Counter(nums) which counts the frequency of each element in the nums

**Space Complexity** 

array. Constructing this frequency counter takes O(N) time, where N is the number of elements in nums.

- The space complexity of the function is the additional space required by the data structures used: Frequency Counter: The Counter will at most store N key-value pairs if all elements in nums are unique, which takes O(N)
- space. **Heap:** The heap size is maintained at k, so the space required for the heap is O(k).

Therefore, the overall space complexity is O(N + k). However, in most scenarios, we expect k to be much smaller than N,

therefore the space complexity is often expressed as O(N).