347. Top k Frequent Elements (/problems/top-k-frequent-elements/)

May 22, 2020 | 49.9K views

Average Rating: 4.51 (74 votes)

Given a non-empty array of integers, return the ***k*** most frequent elements.

Example 1:

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

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Note:

- You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
- Your algorithm's time complexity **must be** better than O(*n* log *n*), where *n* is the array's size.
- It's guaranteed that the answer is unique, in other words the set of the top k frequent elements is unique.
- You can return the answer in any order.

Solution

Approach 1: Heap

Let's start from the simple heap (https://en.wikipedia.org/wiki/Heap_(data_structure)) approach with $\mathcal{O}(N \log k)$ time complexity. To ensure that $\mathcal{O}(N \log k)$ is always less than $\mathcal{O}(N \log N)$, the particular case $k = N$ could be considered separately and solved in $\mathcal{O}(N)$ time.

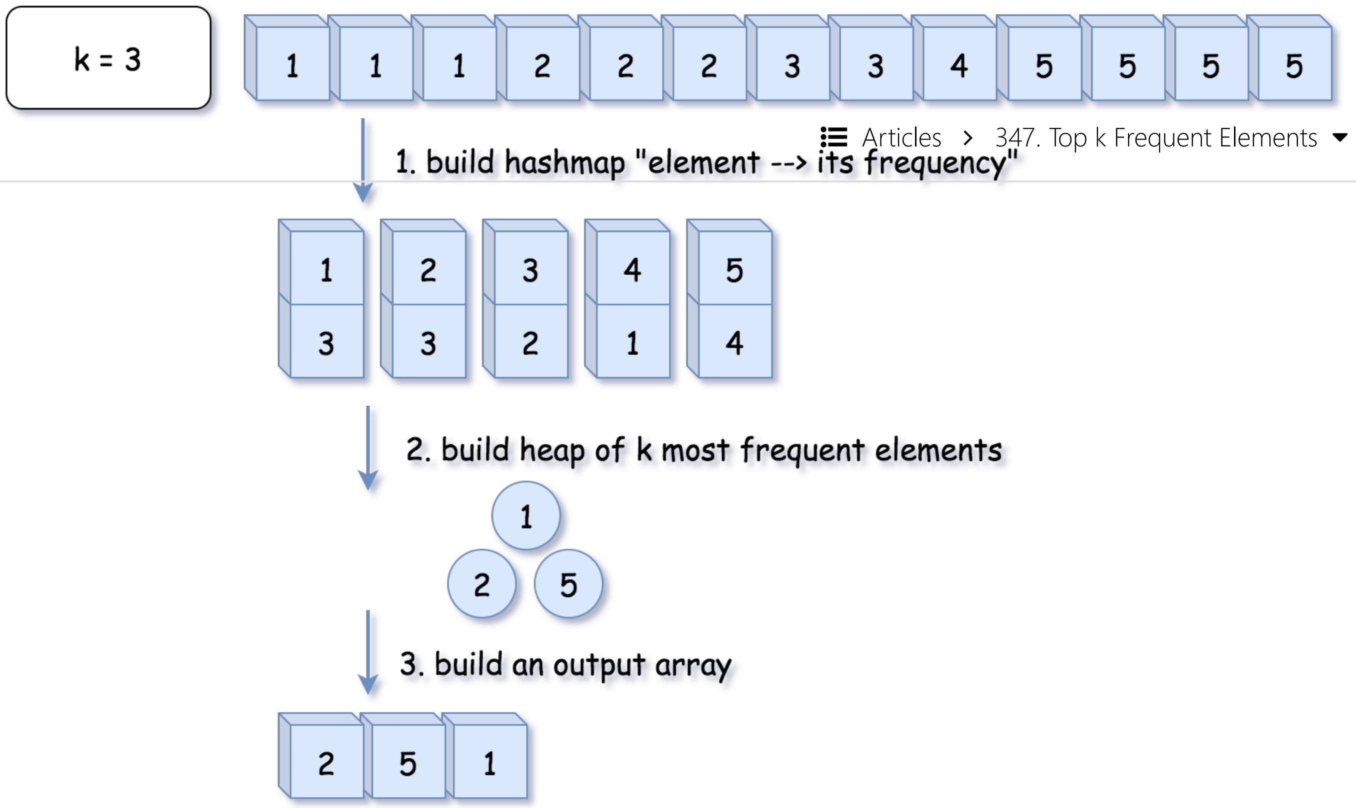
Algorithm

- The first step is to build a hash map `element -> its frequency`. In Java, we use the data structure `HashMap`. Python provides dictionary subclass `Counter` to initialize the hash map we need directly from the input array.
  This step takes $\mathcal{O}(N)$ time where `N` is a number of elements in the list.

- The second step is to build a heap of *size k using N elements*. To add the first `k` elements takes a linear time $\mathcal{O}(k)$ in the average case, and $\mathcal{O}(\log 1 + \log 2 + ... + \log k) = \mathcal{O}(logk!) = \mathcal{O}(k \log k)$ in the worst case. It's equivalent to heapify implementation in Python (https://hg.python.org/cpython/file/2.7/Lib/heapq.py#l16). After the first `k` elements we start to push and pop at each step, `N - k` steps in total. The time complexity of heap push/pop is $\mathcal{O}(\log k)$ and we do it `N - k` times that means $\mathcal{O}((N - k) \log k)$ time complexity. Adding both parts up, we get $\mathcal{O}(N \log k)$ time complexity for the second step.

- The third and the last step is to convert the heap into an output array. That could be done in $\mathcal{O}(k \log k)$ time.

In Python, library `heapq` provides a method `nlargest`, which combines the last two steps under the hood (https://hg.python.org/cpython/file/2.7/Lib/heapq.py#l203) and has the same $\mathcal{O}(N \log k)$ time complexity.


k = 3
1 1 1 2 2 2 3 3 4 5 5 5 5
1. build hashmap "element --> its frequency"
1 2 3 4 5
3 3 2 1 4
2. build heap of k most frequent elements
1
2 5
3. build an output array
2 5 1

Implementation

Java Python Copy

```
from collections import Counter
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # O(1) time
        if k == len(nums):
            return nums

        # 1. build hash map : character and how often it appears
        # O(N) time
        count = Counter(nums)
        # 2-3. build heap of top k frequent elements and
        # convert it into an output array
        # O(N log k) time
        return heapq.nlargest(k, count.keys(), key=count.get)
```

Complexity Analysis

- Time complexity : $\mathcal{O}(N \log k)$ if $k < N$ and $\mathcal{O}(N)$ in the particular case of $N = k$. That ensures time complexity to be better than $\mathcal{O}(N \log N)$.
- Space complexity : $\mathcal{O}(N + k)$ to store the hash map with not more N elements and a heap with k elements.

Approach 2: Quickselect

Hoare's selection algorithm

Quickselect is a textbook algorthm (https://en.wikipedia.org/wiki/Quickselect) typically used to solve the problems "find k *th* something": k *th* smallest, k *th* largest, k *th* most frequent, k *th* less frequent, etc. Like quicksort, quickselect was developed by Tony Hoare (https://en.wikipedia.org/wiki/Tony_Hoare), and also known as *Hoare's selection algorithm*.

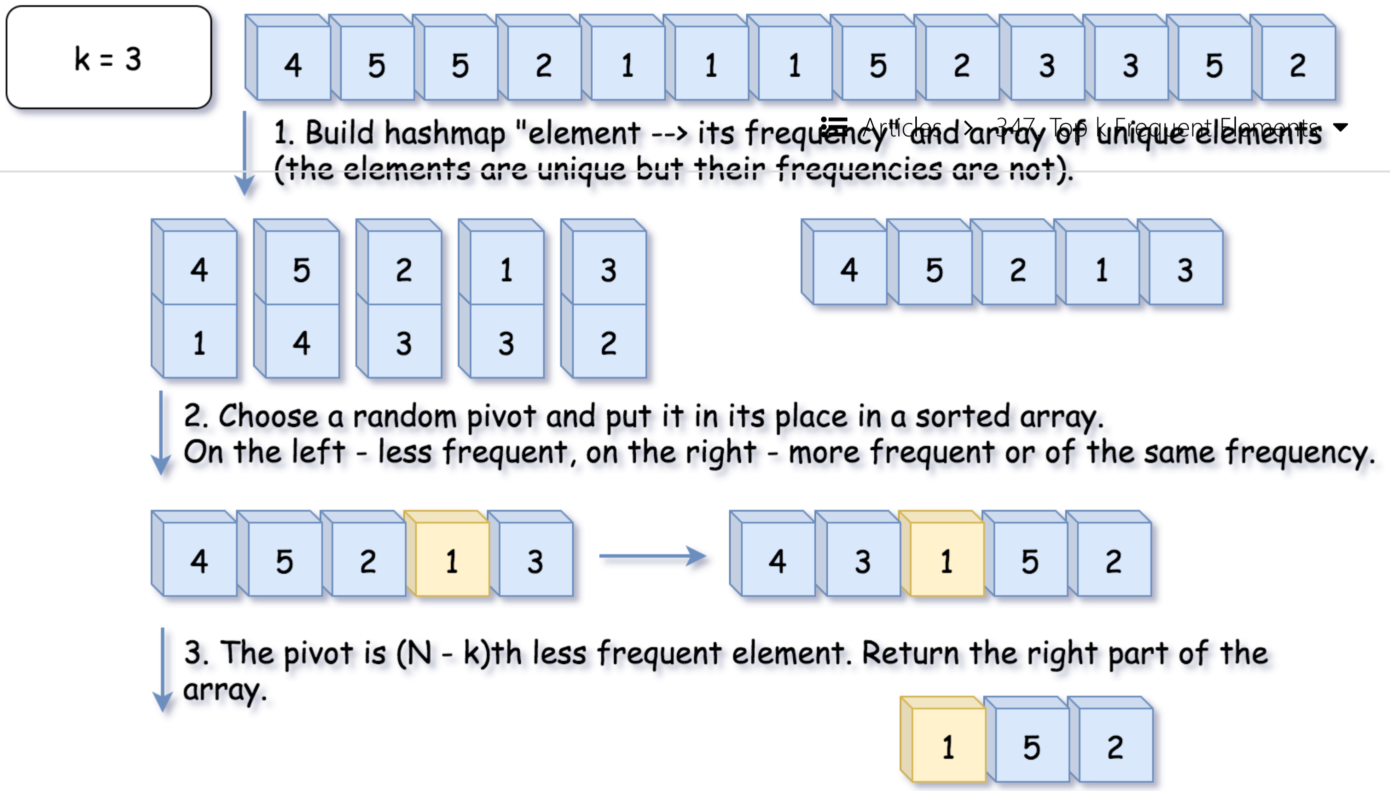
It has $\mathcal{O}(N)$ *average* time complexity and widely used in practice. It worth to note that its worth case time complexity is $\mathcal{O}(N^2)$, although the probability of this worst-case is negligible.

The approach is the same as for quicksort.

> One chooses a pivot and defines its position in a sorted array in a linear time using so-called *partition algorithm*.

As an output, we have an array where the pivot is on its perfect position in the ascending sorted array, sorted by the frequency. All elements on the left of the pivot are less frequent than the pivot, and all elements on the right are more frequent or have the same frequency.

Hence the array is now split into two parts. If by chance our pivot element took `N - k` *th* final position, then k elements on the right are these top k frequent we're looking for. If not, we can choose one more pivot and place it in its perfect position.

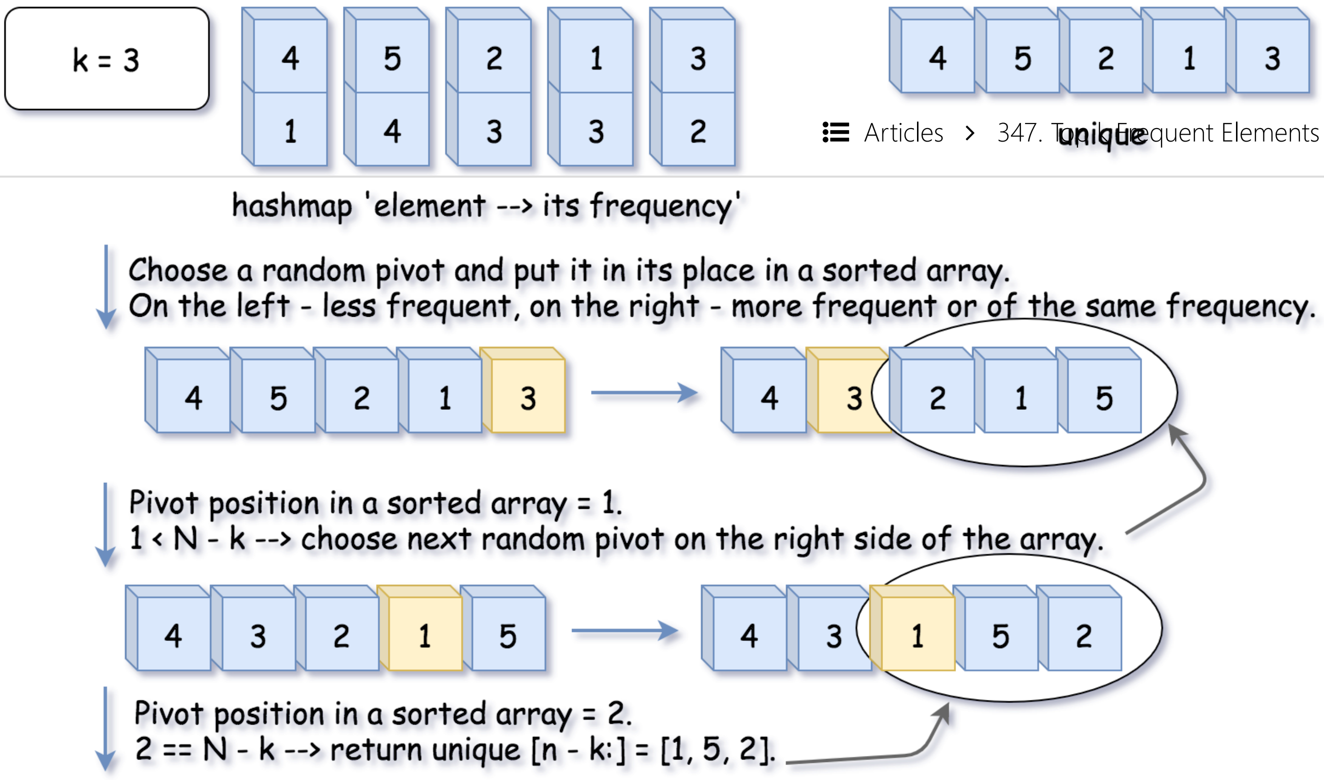

k = 3
4 5 5 2 1 1 1 5 2 3 3 5 2
1. Build hashmap "element --> its frequency" and array of unique elements
(the elements are unique but their frequencies are not).
4 5 2 1 3
1 4 3 3 2
4 5 2 1 3
2. Choose a random pivot and put it in its place in a sorted array.
On the left - less frequent, on the right - more frequent or of the same frequency.
4 5 2 1 3
4 3 1 5 2
3. The pivot is (N - k)th less frequent element. Return the right part of the array.
1 5 2

If that were a quicksort algorithm, one would have to process both parts of the array. That would result in $\mathcal{O}(N \log N)$ time complexity. In this case, there is no need to deal with both parts since one knows in which part to search for `N - k` *th* less frequent element, and that reduces the average time complexity to $\mathcal{O}(N)$.

Algorithm

The algorithm is quite straightforward :

- Build a hash map `element -> its frequency` and convert its keys into the array `unique` of unique elements. Note that elements are unique, but their frequencies are *not*. That means we need a partition algorithm that works fine with *duplicates*.
- Work with `unique` array. Use a partition scheme (please check the next section) to place the pivot into its perfect position `pivot_index` in the sorted array, move less frequent elements to the left of pivot, and more frequent or of the same frequency - to the right.
- Compare `pivot_index` and `N - k`.
    - If `pivot_index == N - k`, the pivot is `N - k` *th* most frequent element, and all elements on the right are more frequent or of the same frequency. Return these top k frequent elements.
    - Otherwise, choose the side of the array to proceed recursively.


k = 3
hashmap 'element --> its frequency'
unique
Articles > 347. Top k Frequent Elements
Choose a random pivot and put it in its place in a sorted array.
On the left - less frequent, on the right - more frequent or of the same frequency.
Pivot position in a sorted array = 1.
1 < N - k --> choose next random pivot on the right side of the array.
Pivot position in a sorted array = 2.
2 == N - k --> return unique [n - k:] = [1, 5, 2].

Hoare's Partition vs Lomuto's Partition

There is a zoo of partition algorithms. The most simple one is Lomuto's Partition Scheme (https://en.wikipedia.org/wiki/Quicksort#Lomuto_partition_scheme).

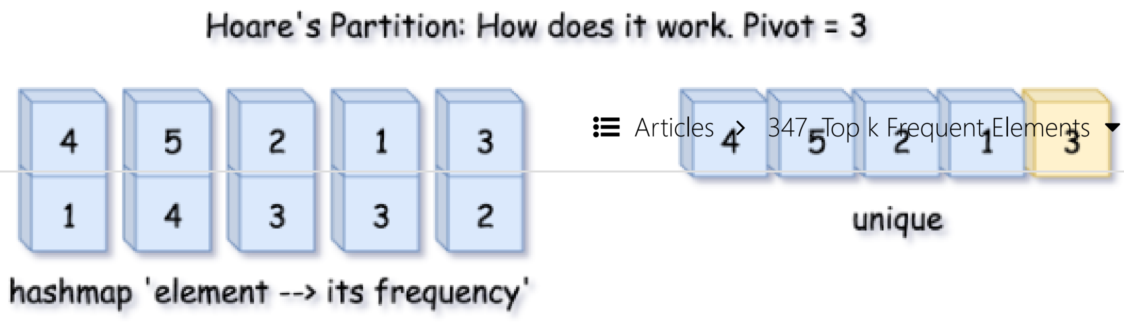
> The drawback of Lomuto's partition is it fails with duplicates.

Here we work with an array of unique elements, but they are compared by frequencies, which are *not unique*. That's why we choose *Hoare's Partition* here.

> Hoare's partition is more efficient than Lomuto's partition because it does three times fewer swaps on average, and creates efficient partitions even when all values are equal.

Here is how it works:

- Move pivot at the end of the array using swap.
- Set the pointer at the beginning of the array `store_index = left`.
- Iterate over the array and move all less frequent elements to the left `swap(store_index, i)`. Move `store_index` one step to the right after each swap.
- Move the pivot to its final place, and return this index.


Hoare's Partition: How does it work. Pivot = 3
4 5 2 1 3
1 4 3 3 2
hashmap 'element --> its frequency'
4 5 2 1 3
unique
Articles > 347. Top k Frequent Elements

Java Python3 Copy

```
public int partition(int left, int right, int pivot_index) {
    int pivot_frequency = count.get(unique[pivot_index]);
    // 1. move pivot to end
    swap(pivot_index, right);
    int store_index = left;

    // 2. move all less frequent elements to the left
    for (int i = left; i <= right; i++) {
        if (count.get(unique[i]) < pivot_frequency) {
            swap(store_index, i);
            store_index++;
        }
    }

    // 3. move pivot to its final place
    swap(store_index, right);

    return store_index;
}
```

Implementation

Here is a total algorithm implementation.

Java Python Copy

Articles > 347. Top k Frequent Elements

```python
from collections import Counter
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        count = Counter(nums)
        unique = list(count.keys())

        def partition(left, right, pivot_index) -> int:
            pivot_frequency = count[unique[pivot_index]]
            # 1. move pivot to end
            unique[pivot_index], unique[right] = unique[right], unique[pivot_index]

            # 2. move all less frequent elements to the left
            store_index = left
            for i in range(left, right):
                if count[unique[i]] < pivot_frequency:
                    unique[store_index], unique[i] = unique[i], unique[store_index]
                    store_index += 1

            # 3. move pivot to its final place
            unique[right], unique[store_index] = unique[store_index], unique[right]

            return store_index

        def quickselect(left, right, k_smallest) -> None:
            """
            Sort a list within left..right till kth less frequent element
            takes its place.
```

Complexity Analysis

- Time complexity: $\mathcal{O}(N)$ in the average case, $\mathcal{O}(N^2)$ in the worst case. Please refer to this card for the good detailed explanation of Master Theorem (https://leetcode.com/explore/learn/card/recursion-ii/470/divide-and-conquer/2871/). Master Theorem helps to get an average complexity by writing the algorithm cost as $T(N) = aT(N/b) + f(N)$. Here we have an example of Master Theorem case III: $T(N) = T\left(\frac{N}{2}\right) + N$, that results in $\mathcal{O}(N)$ time complexity. That's the case of random pivots.

  In the worst-case of constantly bad chosen pivots, the problem is not divided by half at each step, it becomes just one element less, that leads to $\mathcal{O}(N^2)$ time complexity. It happens, for example, if at each step you choose the pivot not randomly, but take the rightmost element. For the random pivot choice the probability of having such a worst-case is negligibly small.

- Space complexity: up to $\mathcal{O}(N)$ to store hash map and array of unique elements.

Further Discussion: Could We Do Worst-Case Linear Time?

In theory, we could, the algorithm is called Median of Medians (https://en.wikipedia.org/wiki/Median_of_medians).

This method is never used in practice because of two drawbacks:

- It's *outperformer*. Yes, it works in a linear time αN, but the constant α is so large that in practice it often works even slower than N^2.

- It doesn't work with duplicates.

Rate this article:

Previous (/articles/is-subsequence/)

Next (/articles/find-all-duplicates-in-an-array/)

Comments: 28

Sort By

Type comment here... (Markdown is supported)

Preview Post

xiaoliu3 (/xiaoliu3) ★ 28 May 31, 2020 7:51 AM

so clear

21 Share Reply

DBabichev (/dbabichev) ★ 2074 June 16, 2020 12:02 PM

There is in fact `O(n)` time and space solutoin, using **bucket sort** idea: in fact, not frequencies can be more than `n`. I think this solution is simpler and better than overcomplicated QuickSelect solution.

```
def topKFrequent(self, nums, k):
```

Read More

32 Share Reply

SHOW 7 REPLIES

denis19 (/denis19) ★ 24 June 11, 2020 11:21 AM

Solid explanation. It took me some time to fully understand what's going on, but since this algorithm is a staple for k-selection problems, I think it's worth the headache.

10 Share Reply

SHOW 3 REPLIES

aakash9518 (/aakash9518) ★ 6 June 3, 2020 10:26 PM

nice

6 Share Reply Articles › 347. Top k Frequent Elements

harman8911 (/harman8911) 16 June 5, 2020 4:48 AM

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered map<int,int> mp;
```

Read More

3 Share Reply

intern (/intern) 5 June 9, 2020 12:20 PM

You are not using Hoare's partition here - you are using Lomuto's. Hoare's partition fails with duplicates, Lomuto's doesn't.

5 Share Reply

SHOW 4 REPLIES

e3p (/e3p) 5 July 13, 2020 10:10 AM

A disadvantage of the quickselect algorithm (as it's currently implemented) is that it won't return the top **K** elements in sorted order, right?

2 Share Reply

SHOW 1 REPLY

edmat7 (/edmat7) 18 July 3, 2020 11:05 PM

For Approach 1, when k = N, the time complexity is O(1) not O(N) since you just return the input array when k=N

1 Share Reply

SHOW 2 REPLIES

dkhatri (/dkhatri) 10 July 13, 2020 7:51 AM

Really nice explanation, thanks for sharing.
Another solution can be using two hashmap
1.key_count_map `<key, count>`
2. count_to_keys_map `<count, set<keys>>`
Then processing `count_to_keys_map` in decreasing order of frequency until you find `k` elements

Read More

0 Share Reply

carlos6125 (/carlos6125) 15 June 7, 2020 9:58 PM

Shouldn't approach 1 be O(N). I thought heapifying/building a heap was linear?

0 Share Reply

SHOW 4 REPLIES

‹ 1 2 3 ›

Articles › 347. Top k Frequent Elements