Problem 347: Top K Frequent Elements

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given an integer array

nums

and an integer

k

, return

the

k

most frequent elements

. You may return the answer in

any order

.

Example 1:

Input:

nums = [1,1,1,2,2,3], k = 2

Output:

[1,2]

Example 2:

Input:

nums = [1], k = 1

Output:

[1]

Example 3:

Input:

nums = [1,2,1,2,1,2,3,1,3,2], k = 2

Output:

[1,2]

Constraints:

1 <= nums.length <= 10

5

-10

4

<= nums[i] <= 10

4

k

is in the range

[1, the number of unique elements in the array]

.

It is

guaranteed

that the answer is

unique

.

Follow up:

Your algorithm's time complexity must be better than

O(n log n)

, where n is the array's size.

Code Snippets

C++:

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

}
};
```

Java:

```
class Solution {
public int[] topKFrequent(int[] nums, int k) {

}
}
```

Python3:

```
class Solution:
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
```

Python:

```
class Solution(object):
def topKFrequent(self, nums, k):
"""
:type nums: List[int]
:type k: int
:rtype: List[int]
"""
```

JavaScript:

```
/**
* @param {number[]} nums
* @param {number} k
* @return {number[]}
*/
var topKFrequent = function(nums, k) {

};
```

TypeScript:

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

};
```

C#:

```
public class Solution {
public int[] TopKFrequent(int[] nums, int k) {

}
}
```

C:

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* topKFrequent(int* nums, int numsSize, int k, int* returnSize) {

}
```

Go:

```
func topKFrequent(nums []int, k int) []int {

}
```

Kotlin:

```
class Solution {
fun topKFrequent(nums: IntArray, k: Int): IntArray {

}
}
```

Swift:

```
class Solution {
func topKFrequent(_ nums: [Int], _ k: Int) -> [Int] {

}
}
```

Rust:

```
impl Solution {
pub fn top_k_frequent(nums: Vec<i32>, k: i32) -> Vec<i32> {

}
}
```

Ruby:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer[]}
def top_k_frequent(nums, k)

end
```

PHP:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer[]
*/
function topKFrequent($nums, $k) {

}
}
```

Dart:

```
class Solution {
List<int> topKFrequent(List<int> nums, int k) {

}
}
```

Scala:

```
object Solution {
def topKFrequent(nums: Array[Int], k: Int): Array[Int] = {

}
}
```

Elixir:

```
defmodule Solution do
@spec top_k_frequent(nums :: [integer], k :: integer) :: [integer]
```

```
def top_k_frequent(nums, k) do

end
end
```

Erlang:

```
-spec top_k_frequent(Nums :: [integer()], K :: integer()) -> [integer()].
top_k_frequent(Nums, K) ->
.
```

Racket:

```
(define/contract (top-k-frequent nums k)
(-> (listof exact-integer?) exact-integer? (listof exact-integer?))
)
```

Solutions

C++ Solution:

```
/*
* Problem: Top K Frequent Elements
* Difficulty: Medium
* Tags: array, hash, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public:
vector<int> topKFrequent(vector<int>& nums, int k) {

}
};
```

Java Solution:

```
/**
* Problem: Top K Frequent Elements
* Difficulty: Medium
* Tags: array, hash, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class Solution {
public int[] topKFrequent(int[] nums, int k) {

}
}
```

Python3 Solution:

```
"""
Problem: Top K Frequent Elements
Difficulty: Medium
Tags: array, hash, sort, queue, heap

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) for hash map
"""

class Solution:
def topKFrequent(self, nums: List[int], k: int) -> List[int]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class Solution(object):
def topKFrequent(self, nums, k):
"""
:type nums: List[int]
:type k: int
:rtype: List[int]
"""
```

JavaScript Solution:

```
/**
* Problem: Top K Frequent Elements
* Difficulty: Medium
* Tags: array, hash, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number[]} nums
* @param {number} k
* @return {number[]}
*/
var topKFrequent = function(nums, k) {

};
```

TypeScript Solution:

```
/**
* Problem: Top K Frequent Elements
* Difficulty: Medium
* Tags: array, hash, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

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

};
```

C# Solution:

```
/*
* Problem: Top K Frequent Elements
* Difficulty: Medium
* Tags: array, hash, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class Solution {
public int[] TopKFrequent(int[] nums, int k) {

}
}
```

C Solution:

```
/*
* Problem: Top K Frequent Elements
* Difficulty: Medium
* Tags: array, hash, sort, queue, heap
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* topKFrequent(int* nums, int numsSize, int k, int* returnSize) {

}
```

Go Solution:

```
// Problem: Top K Frequent Elements
// Difficulty: Medium
// Tags: array, hash, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map
```

```
func topKFrequent(nums []int, k int) []int {

}
```

Kotlin Solution:

```
class Solution {
fun topKFrequent(nums: IntArray, k: Int): IntArray {

}
}
```

Swift Solution:

```
class Solution {
func topKFrequent(_ nums: [Int], _ k: Int) -> [Int] {

}
}
```

Rust Solution:

```
// Problem: Top K Frequent Elements
// Difficulty: Medium
// Tags: array, hash, sort, queue, heap
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

impl Solution {
pub fn top_k_frequent(nums: Vec<i32>, k: i32) -> Vec<i32> {

}
}
```

Ruby Solution:

```
# @param {Integer[]} nums
# @param {Integer} k
# @return {Integer[]}
def top_k_frequent(nums, k)

end
```

PHP Solution:

```
class Solution {

/**
* @param Integer[] $nums
* @param Integer $k
* @return Integer[]
*/
function topKFrequent($nums, $k) {

}
}
```

Dart Solution:

```
class Solution {
List<int> topKFrequent(List<int> nums, int k) {

}
}
```

Scala Solution:

```
object Solution {
def topKFrequent(nums: Array[Int], k: Int): Array[Int] = {

}
}
```

Elixir Solution:

```
defmodule Solution do
@spec top_k_frequent(nums :: [integer], k :: integer) :: [integer]
def top_k_frequent(nums, k) do
```

```
end
end
```

Erlang Solution:

```
-spec top_k_frequent(Nums :: [integer()], K :: integer()) -> [integer()].
top_k_frequent(Nums, K) ->
.
```

Racket Solution:

```
(define/contract (top-k-frequent nums k)
(-> (listof exact-integer?) exact-integer? (listof exact-integer?))
)
```