

# HEAP / PRIORITY QUEUE

# Heap

- **Heap** is a complete binary tree that satisfy the heap property (max or min)
- **Min heap**: root node contains the minimum value
- **Max heap**: root node contains the maximum value

Min Heap



Max Heap



# Heap in C++

## Two main ways to implement:

### 1. Using **std::make\_heap** from **<algorithm>**

```
std::make_heap(RandomIt first, RandomIt last)
```

```
std::push_heap(RandomIt first, RandomIt last)
```

```
std::pop_heap(RandomIt first, RandomIt last)
```

```
std::sort_heap(RandomIt first, RandomIt last)
```

### 2. Using **std::priority\_queue** from **<queue>** **(recommended)**

```
std::priority_queue<T, Container, Compare>
```

# Heap in C++ – std::priority\_queue example

## Min heap

```
std::priority_queue<int, std::vector<int>, std::greater<int>>>
```

## Max heap

```
std::priority_queue<int> or
```

```
std::priority_queue<int, std::vector<int>, std::less<int>>>
```

```
// Min heap
std::priority_queue<int, std::vector<int>, std::greater<int>>> minHeap;

minHeap.push(3);
minHeap.push(6);
minHeap.push(4);
// remove top element (3)
minHeap.pop();
// root node (top) is now 4
std::cout << minHeap.top();
```

# Problem – Kth Largest Element in an Array

Medium

<https://leetcode.com/problems/kth-largest-element-in-an-array>

## Problem

Given an integer array `nums` and an integer `k`, return the  $k^{\text{th}}$  largest element in the array. Note that it is the  $k^{\text{th}}$  largest element in the sorted order, not the  $k^{\text{th}}$  distinct element.

## Example 1

Input: `nums = [3,2,1,5,6,4]`, `k = 2`

Output: 5

## Example 2

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`

Output: 4

Although this problem is classified as “medium”, in my opinion it should be classified as “easy”

# Solution 1 – Kth Largest Element in an Array

Medium

<https://leetcode.com/problems/kth-largest-element-in-an-array>

// SOLUTION 1

```
int findKthLargest(vector<int>& nums, int k) {
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
    for (const auto& num : nums) {
        if (minHeap.size() < k) {
            minHeap.push(num);
        } else if (num > minHeap.top()) {
            minHeap.pop();
            minHeap.push(num);
        }
    }
    return minHeap.top();
}
```

# Solution 2 – Kth Largest Element in an Array

Medium

<https://leetcode.com/problems/kth-largest-element-in-an-array>

```
// SOLUTION 2 - Simpler approach
```

```
int findKthLargest(vector<int>& nums, int k) {  
    // min heap: minimum values will be always at the top  
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;  
    for (const auto& num : nums) {  
        // push each num to the heap  
        minHeap.push(num);  
        // we need the kth largest element only, so once after pushing more than k  
        // elements, remove the smallest one (the top)  
        if (minHeap.size() > k) {  
            minHeap.pop();  
        }  
    }  
    return minHeap.top();  
}
```

# Problem – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

## Problem

- You are given an **array** of numbers and an **integer**  $k$
- Return an array with the  **$k$**  most frequent elements

Example

**Input:**

$\text{nums} = [1, 1, 1, 2, 2, 3], k = 2$

**Output:**

$[1, 2]$



# Solution – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

## Solution (1) - hashmap + array sort

- Go over the array, count the numbers and store them in an *unordered\_map*

### Example:

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

`freq[1] = 3`

`freq[2] = 2`

`...`

- Go over the *unordered\_map*, add to an array and sort descending
- Create another array adding the **k** first elements and return

# Code – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

**Code (1)** Time:  $O(n \log n)$  Space:  $O(n)$

```
vector<int> topKFrequent(vector<int>& nums, int k) {  
    // 1. Create the number's frequency map  
    // O(n)  
    unordered_map<int, int> freq;  
    for (const auto& num : nums) {  
        freq[num] += 1;  
    }  
    // 2. Create an array with the frequencies  
    vector<pair<int, int>> freqVec(freq.begin(), freq.end());  
    // 3. Sort by the frequency  $O(n \log n)$   
    sort(freqVec.begin(), freqVec.end(), [](auto& a, auto& b) {  
        return a.second > b.second;  
    });  
    // 4. Create the result with the k first elements  
    // O(k)  
    vector<int> result;  
    for (int i = 0; i < k; ++i) {  
        result.push_back(freqVec[i].first);  
    }  
    return result;  
}
```

# Solution – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

## Solution (2) - hashmap + min heap

- Go over the array, count the numbers and store them in an *unordered\_map*

### Example:

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

`freq[1] = 3`

`freq[2] = 2`

`...`

- Go over the frequencies, add to a min heap. If the size of the heap exceeds **k**, remove the top one (the minimum value)
- Create another array result adding all elements from the heap and return it

# Code – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

**Code (2)** Time:  $O(n \log k)$  Space:  $O(n)$

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    // 1. Create the number's frequency map
    // O(n)
    unordered_map<int, int> freq;
    for (const auto& num : nums) {
        freq[num] += 1;
    }
    // 2. Create the min heap with priority queue
    // O(n log k)
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> minHeap;
    for (const auto& [num, count] : freq) {
        minHeap.push({count, num});
        if (minHeap.size() > k) minHeap.pop();
    }
    // 3. build the result
    vector<int> result;
    while (!minHeap.empty()) {
        auto num = minHeap.top().second;
        minHeap.pop();
        result.push_back(num);
    }
    return result;
}
```

# Solution – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

## Solution (3) - hashmap + bucket sort

- Go over the array, count the numbers and store them in an *unordered\_map*

### Example:

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

`freq[1] = 3`

`freq[2] = 2`

...

- Create buckets for each frequency and add the corresponding numbers:

`bucket[1] = [3]` → 3 only appears once in nums

`bucket[2] = [2]` → 2 appears twice

`bucket[3] = [1]` → 1 appears three times

- Go over each bucket, add to the result and return it

# Code – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

**Code (3)** Time:  $O(n)$  Space:  $O(n)$

```
vector<int> topKFrequent(vector<int>& nums, int k) {  
    // Create the number's frequency map  
    unordered_map<int, int> freq;  
    for (const auto& num : nums) {  
        freq[num]++;  
    }  
    // create the buckets  
    // e.g. [[1,2,3],[4,5,6]] ...  
    vector<vector<int>> buckets(nums.size() + 1);  
    for (const auto& [num, count] : freq) {  
        buckets[count].push_back(num);  
    }  
    // go over each bucket to build the result  
    vector<int> result;  
    for (int i = buckets.size() - 1; i >= 0; --i) {  
        for (const auto& num : buckets[i]) {  
            result.push_back(num);  
            if (result.size() == k) return result;  
        }  
    }  
    return result;  
}
```

# Problem – 347. Top K Frequent Elements

Medium



LeetCode

[leetcode.com/problems/top-k-frequent-elements](https://leetcode.com/problems/top-k-frequent-elements)

## Some considerations

- Theoretically, bucket sort should be the fastest solution  $O(n) < O(n \log k)$
- In practice, min heap end up being faster:
  - fewer allocations: priority\_queue stores flat pairs rather than inner vectors
  - better cache locality: heap is built over a single array (binary heap)
  - if **k** is small, heap touches fewer elements