# 题目

给一非空的单词列表，返回前 k 个出现次数最多的单词。

返回的答案应该按单词出现频率由高到低排序。如果不同的单词有相同出现频率，按字母顺序排序。

**示例 1：**

**输入:** ["i", "love", "leetcode", "i", "love", "coding"], k = 2

**输出:** ["i", "love"]

**解析:** "i" 和 "love" 为出现次数最多的两个单词，均为2次。

注意，按字母顺序 "i" 在 "love" 之前。

**示例 2：**

**输入:** ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4

**输出:** ["the", "is", "sunny", "day"]

**解析:** "the", "is", "sunny" 和 "day" 是出现次数最多的四个单词，

出现次数依次为 4, 3, 2 和 1 次。

**注意：**

假定 k 总为有效值， 1 ≤ k ≤ 集合元素数。

输入的单词均由小写字母组成。

**扩展练习：**

尝试以 O(n log k) 时间复杂度和 O(n) 空间复杂度解决。

类似题目：Leetcode 347

# 分析

## 方法一：排序

**思路：**

计算每个单词的频率，并使用使用这些频率的自定义排序关系对单词进行排序。然后取前k。

**代码（Java）：**

class Solution {

public List<String> topKFrequent(String[] words, int k) {

Map<String, Integer> count = new HashMap();

for (String word: words) {

count.put(word, count.getOrDefault(word, 0) + 1);

}

List<String> candidates = new ArrayList(count.keySet());

Collections.sort(candidates, (w1, w2) -> count.get(w1).equals(count.get(w2)) ?

w1.compareTo(w2) : count.get(w2) - count.get(w1));

return candidates.subList(0, k);

**复杂度：**

时间复杂度：O(NlogN)。其中N是words的长度。我们用O(N)时间计算每个单词的频率，然后用O(NlogN) 时间对给定的单词进行排序。

空间复杂度：O(N)，用来存放候答案的地方。

## 方法二：堆/优先队列/哈希表

**思路：**

计算每个单词的频率，然后将其添加到存储到大小为k的小根堆中。它将频率最小的候选项放在堆的顶部。最后，我们从堆中弹出最多k次，并反转结果，就可以得到前k个高频单词。

在Python中，我们使用heapq/heapify，它可以在线性时间内将列表转换为堆，从而简化了我们的工作。

**代码：**

class Solution {

public:

vector<string> topKFrequent(vector<string>& words, int k) {

struct compare{

bool operator() (std::pair<string,int> &p1, std::pair<string,int> &p2){

if(p1.second == p2.second)

return p1.first > p2.first;

return p1.second < p2.second;

}

};

unordered\_map<string,int> mp;

for(auto w : words)

mp[w]++;

std::priority\_queue<std::pair<string,int>,std::vector<std::pair<string,int>>, compare> queue;

for(auto m: mp)

queue.push(make\_pair(m.first,m.second));

std::vector<string> s;

while(k--){

s.push\_back(queue.top().first);

queue.pop();

}

return s;

}

};

或：

class Solution {

public:

    vector<string> topKFrequent(vector<string>& words, int k) {

        struct compare{

            bool operator() (std::pair<string,int> &p1, std::pair<string,int> &p2){

            if(p1.second == p2.second)

                return p1.first > p2.first;

            return p1.second < p2.second;

            }

        };

        std::priority\_queue<std::pair<string,int>,std::vector<std::pair<string,int> >, compare> queue;

        unordered\_map<string,int> mp; //哈希表

        for(auto w : words)

            mp[w]++; //对应字符的个数+1

/\*这是采用C++11的auto遍历，或采用传统的迭代器方式赋值：

for(int i = 0;i<words.size();i++){

mp[words[i]]++;

}

\*/

        for(auto m: mp)

            queue.push(make\_pair(m.first,m.second));

/\*

for(auto it = mp.begin();it != mp.end();it++){

queue.push({it->first,it->second});

}

\*/

        std::vector<string> s;

        while(k--){

            s.push\_back(queue.top().first);

            queue.pop();

        }

        return s;

    }

};

**复杂度：**

时间复杂度：O(Nlogk)。其中N是words的长度。我们用O(N)的时间计算每个单词的频率，然后将N个单词添加到堆中，添加每个单词的时间为O(logk)。最后，我们从堆中弹出最多k次。因为k≤N的值，总共是O(Nlogk)。

空间复杂度：O(N)，用于存储我们计数的空间。