# 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。

**类似题目：**Leetcode 692

# 分析

有两种基本的思路：

## 方法一：排序

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。

## 方法二：堆/优先队列/哈希表

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。

在构造优先队列/堆的时候，可以采用如下的方法：

1、静态函数

static bool compare(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;

2、仿函数：重载运算符

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;

}

};

3、Lambda表达式

注：这种其实就是第k大/小元素的变式，只不过原来构造优先队列的时候只需要存储节点数值和比较大小即可，现在需要存储pair，且比较函数需要自行编写。

**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;

            }

        };

        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;

    }

};

或：

typedef pair<int, int> IIPair;

struct cmp {

bool operator()(const IIPair &left, const IIPair &right) const

{

return left.second > right.second;

}

};

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

unordered\_map<int, int> mp;

for (auto i : nums) {

mp[i]++;

}

priority\_queue<IIPair, vector<IIPair>, cmp> q; // 创建一个小顶堆

for (auto item : mp) {

if (q.size() < k) {

q.push(item);

} else if (item.second > q.top().second) {

q.pop();

q.push(item);

}

// 新的元素频次小于堆顶元素频次的元素不处理。

}

vector<int> ret(q.size(), 0);

while (!q.empty()) {

ret[q.size()-1] = q.top().first;

q.pop();

}

return ret;

}

**复杂度：**

**时间复杂度**

统计频率，需要遍历一遍数组，所需要的时间复杂度为O(n), hashmap的插入、查找的时间复杂度为O(1), 所以统计频率过程时间复杂度为O(n)。

遍历hashmap的时间复杂度时间复杂度O(n)，维护元素个数为k的小顶堆时间复杂度为O(logk)， 这个过程时间复杂度为O(nlogk)。

把优先队列转换为数组时间复杂度为O(k)。

综上所述，整体时间复杂度为 O(nlogk)。

**空间复杂度**

创建hashmap，最坏情况，当没有相同元素时，复杂度为O(n)

小顶堆空间复杂度为O(k)

保存频率前k的元素数组，复杂度O(k)

综上，空间复杂度为O(n)

## 方法三：快排思想