**Apple Interview**

Introduction

Top interview questions asked by Apple as voted by the community.

We compiled this list thoroughly so you can save time and get well-prepared for an Apple interview.

Completing this card should give you a good idea for the type of questions you would encounter in your Apple interview.

Arrays and Strings

 Two Sum – Array - HashTable

 Longest Substring Without Repeating Characters – HashTable

 String to Integer (atoi) - EZ

 Integer to Roman - AMZ

 Roman to Integer - EZ

 3Sum - Medium

 3Sum Closest - AMZ

 4Sum – HashTable/Hard

 Group Anagrams – HashTable/Medium

 Spiral Matrix - Array

 Minimum Window Substring – Hard/….

 Valid Palindrome -EZ

**Majority Element II**

 Product of Array Except Self - Hard

 Missing Number -EZ

 First Unique Character in a String - EZ

 Subarray Sum Equals K - FB

 Squares of a Sorted Array – Array 101

 Valid Parentheses – Q&S

 Trapping Rain Water - Hard

**Sparse Matrix Multiplication**

Linked Lists

 Add Two Numbers - LKList

 Merge Two Sorted Lists - LKList

 Reverse Linked List - LKlist

Trees and Graphs

 Same Tree – Recursion 2

 Maximum Depth of Binary Tree - EZ

 Clone Graph – Q&S

 Number of Islands – Q&S Medium

 Lowest Common Ancestor of a Binary Tree – Binary Tree

 Longest Increasing Path in a Matrix - Hard

 Diameter of Binary Tree -Google/FB

Recursion

 Letter Combinations of a Phone Number - Medium

 Generate Parentheses - Medium

**Combination Sum**

 Permutations – Medium/FB

 Subsets - Medium

 Word Search - Medium

Sorting and Searching

 Median of Two Sorted Arrays - Hard

 Search in Rotated Sorted Array - Medium

 Merge Intervals - Medium

 Sort Colors - Medium

 Valid Anagram - EZ

 Intersection of Two Arrays – HashTable/FB

 Intersection of Two Arrays II – HashTable/EZ/FB

**Top K Frequent Words**

 K Closest Points to Origin -Google

Dynamic Programming

 Longest Palindromic Substring - Medium

 Regular Expression Matching - Hard

 Maximum Subarray - EZ

 Best Time to Buy and Sell Stock - EZ

 Word Break - Hard

Design

 LRU Cache - Hard

 Min Stack - EZ

 Flatten Nested List Iterator - Hard

 Insert Delete GetRandom O(1) - HashTable

Others

 Reverse Integer -EZ

 Valid Sudoku - EZ

**Combine Two Tables**

**Rank Scores**

 Happy Number - Medium

 Fizz Buzz - EZ

 Jewels and Stones – HashTable

**Majority Element II**

Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

**Follow-up:**Could you solve the problem in linear time and in O(1) space?

**Example 1:**

**Input:** nums = [3,2,3]

**Output:** [3]

**Example 2:**

**Input:** nums = [1]

**Output:** [1]

**Example 3:**

**Input:** nums = [1,2]

**Output:** [1,2]

**Constraints:**

* 1 <= nums.length <= 5 \* 104
* -109 <= nums[i] <= 109

   Hide Hint #1

How many majority elements could it possibly have?  
Do you have a better hint? [Suggest it](mailto:admin@leetcode.com?subject=Hints%20for%20Majority%20Element%20II)!

## Solution

This problem can be approached similarly to [Majority Element](https://leetcode.com/problems/majority-element/). For this problem, two constraints we have to satisfy are linear runtime and constant space. In this article, we will focus on the solution which satisfies both constraints.

#### **Approach 1: Boyer-Moore Voting Algorithm**

**Intuition**

To figure out a O(1)*O*(1) space requirement, we would need to get this simple intuition first. For an array of length n:

* There can be at most **one** majority element which is **more than** ⌊n/2⌋ times.
* There can be at most **two** majority elements which are **more than** ⌊n/3⌋ times.
* There can be at most **three** majority elements which are **more than** ⌊n/4⌋ times.

and so on.

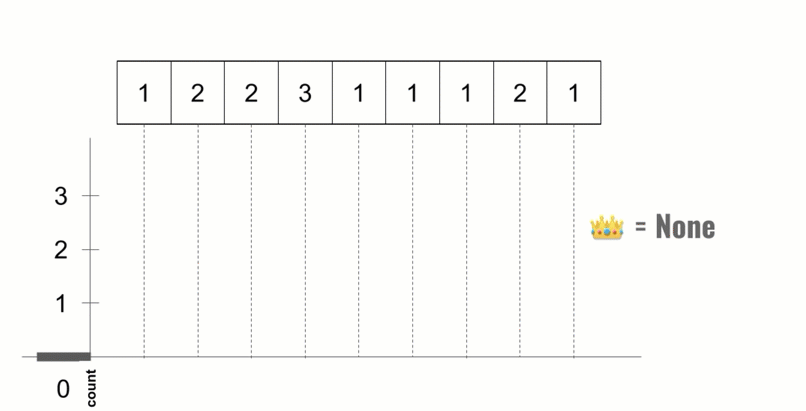
Knowing this can help us understand how we can keep track of majority elements which satisfies O(1)*O*(1) space requirement.

Let's try to get an intuition for the case where we would like to find a majority element which is more than ⌊n/2⌋ times in an array of length n.

The idea is to have two variables, one holding a potential candidate for majority element and a counter to keep track of whether to swap a potential candidate or not. Why can we get away with only two variables? Because there can be at most ***one*** majority element which is more than *⌊n/2⌋* times. Therefore, having only one variable to hold the only potential candidate and one counter is enough.

While scanning the array, the counter is incremented if you encounter an element which is exactly same as the potential candidate but decremented otherwise. When the counter reaches zero, the element which will be encountered next will become the potential candidate. Keep doing this procedure while scanning the array. However, when you have exhausted the array, you have to make sure that the element recorded in the potential candidate variable is the majority element by checking whether it occurs more than ⌊n/2⌋ times in the array. In the original [Majority Element](https://leetcode.com/problems/majority-element/) problem, it is guaranteed that there is a majority element in the array so your implementation can omit the second pass. However, in a general case, you need this second pass since your array can have no majority elements at all!

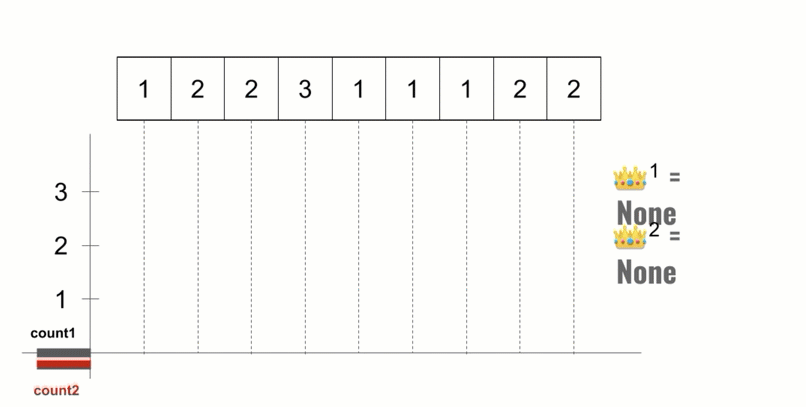
The counter is initialized as 0 and the potential candidate as None at the start of the array.



If an element is truly a majority element, it will stick in the potential candidate variable, no matter how it shows up in the array (i.e. all clustered in the beginning of the array, all clustered near the end of the array, or showing up anywhere in the array), after the whole array has been scanned. Of course, while you are scanning the array, the element might be replaced by another element in the process, but the true majority element will definitely remain as the potential candidate in the end.

Now figuring out the majority elements which show up more than ⌊n/3⌋ times is not that hard anymore. Using the intuition presented in the beginning, we only need four variables: two for holding two potential candidates and two for holding two corresponding counters. Similar to the above case, both candidates are initialized as None in the beginning with their corresponding counters being 0. While going through the array:

* If the current element is equal to one of the potential candidate, the count for that candidate is increased while leaving the count of the other candidate as it is.
* If the counter reaches zero, the candidate associated with that counter will be replaced with the next element **if** the next element is not equal to the other candidate as well.
* Both counters are decremented **only when** the current element is different from both candidates.



|  |
| --- |
| class Solution {  public List < Integer > majorityElement(int[] nums) {  // 1st pass  int count1 = 0;  int count2 = 0;  Integer candidate1 = null;  Integer candidate2 = null;  for (int n: nums) {  if (candidate1 != null && candidate1 == n) {  count1++;  } else if (candidate2 != null && candidate2 == n) {  count2++;  } else if (count1 == 0) {  candidate1 = n;  count1++;  } else if (count2 == 0) {  candidate2 = n;  count2++;  } else {  count1--;  count2--;  }  }  // 2nd pass  List result = new ArrayList <> ();  count1 = 0;  count2 = 0;  for (int n: nums) {  if (candidate1 != null && n == candidate1) count1++;  if (candidate2 != null && n == candidate2) count2++;  }  int n = nums.length;  if (count1 > n/3) result.add(candidate1);  if (count2 > n/3) result.add(candidate2);  return result;  }  } |

**Complexity Analysis**

* Time complexity : O(N)*O*(*N*) where N*N* is the size of nums. We first go through nums looking for first and second potential candidates. We then count the number of occurrences for these two potential candidates in nums. Therefore, our runtime is O(N) + O(N) = O(2N) \approx O(N)*O*(*N*)+*O*(*N*)=*O*(2*N*)≈*O*(*N*).
* Space complexity : O(1)*O*(1) since we only have four variables for holding two potential candidates and two counters. Even the returning array is at most 2 elements.

**Sparse Matrix Multiplication**

Given two [sparse matrices](https://en.wikipedia.org/wiki/Sparse_matrix) **A** and **B**, return the result of **AB**.

You may assume that **A**'s column number is equal to **B**'s row number.

**Example:**

**Input:**

**A** = [

[ 1, 0, 0],

[-1, 0, 3]

]

**B** = [

[ 7, 0, 0 ],

[ 0, 0, 0 ],

[ 0, 0, 1 ]

]

**Output:**

| 1 0 0 | | 7 0 0 | | 7 0 0 |

**AB** = | -1 0 3 | x | 0 0 0 | = | -7 0 3 |

| 0 0 1 |

**Constraints:**

* 1 <= A.length, B.length <= 100
* 1 <= A[i].length, B[i].length <= 100
* -100 <= A[i][j], B[i][j] <= 100

**Combination Sum**

Given an array of **distinct** integers candidates and a target integer target, return a list of all ***unique combinations*** of candidates where the chosen numbers sum to target. You may return the combinations in **any order**.

The **same** number may be chosen from candidates an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

It is **guaranteed** that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

**Example 1:**

**Input:** candidates = [2,3,6,7], target = 7

**Output:** [[2,2,3],[7]]

**Explanation:**

2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

**Example 2:**

**Input:** candidates = [2,3,5], target = 8

**Output:** [[2,2,2,2],[2,3,3],[3,5]]

**Example 3:**

**Input:** candidates = [2], target = 1

**Output:** []

**Example 4:**

**Input:** candidates = [1], target = 1

**Output:** [[1]]

**Example 5:**

**Input:** candidates = [1], target = 2

**Output:** [[1,1]]

**Constraints:**

* 1 <= candidates.length <= 30
* 1 <= candidates[i] <= 200
* All elements of candidates are **distinct**.
* 1 <= target <= 500

## Solution

#### **Overview**

This is one of the problems in the series of combination sum. They all can be solved with the same algorithm, i.e. **backtracking**.

Before tackling this problem, we would recommend one to start with another almost identical problem called [Combination Sum III](https://leetcode.com/problems/combination-sum-iii/), which is arguably easier and one can tweak the solution a bit to solve this problem.

For the sake of this article, we will present the backtracking algorithm. Furthermore, we will list some other problems on LeetCode that one can solve with the same algorithm presented here.

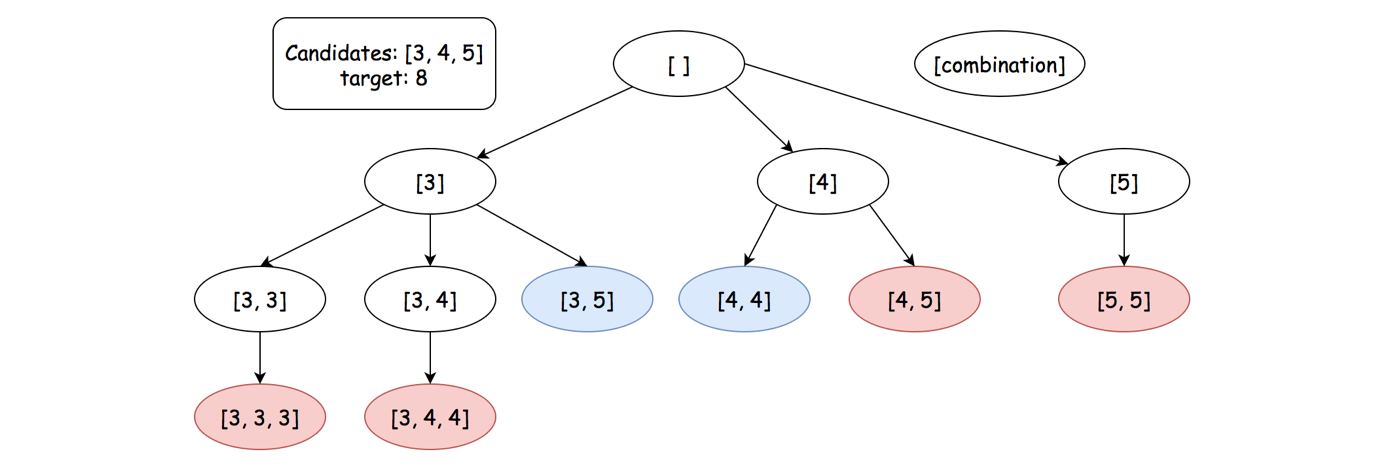
#### **Approach 1: Backtracking**

**Intuition**

As a reminder, [backtracking](https://en.wikipedia.org/wiki/Backtracking) is a general algorithm for finding all (or some) solutions to some computational problems. The idea is that it **incrementally** builds candidates to the solutions, and abandons a candidate ("backtrack") as soon as it determines that this candidate cannot lead to a final solution.

Specifically, to our problem, we could incrementally build the combination, and once we find the current combination is not valid, we backtrack and try another option.

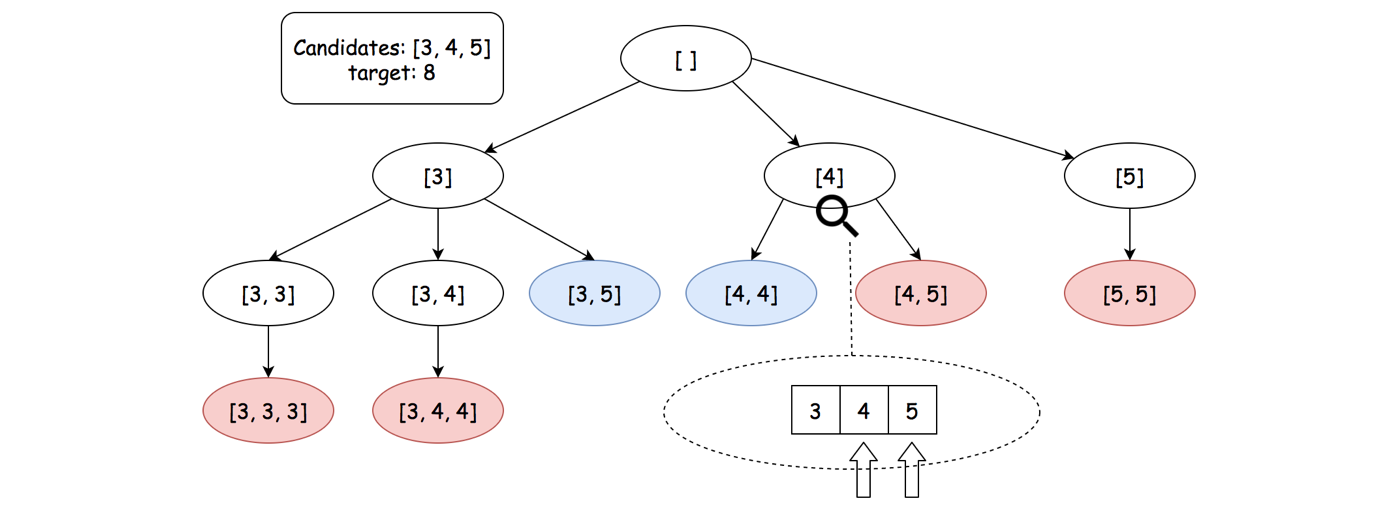
To demonstrate the idea, we showcase how it works with a concrete example in the following graph:



For the given list of candidates [3, 4, 5] and a target sum 8, we start off from empty combination [] as indicated as the root node in the above graph.

* Each node represents an action we take at a step, and within the node we also indicate the combination we build sofar.
* From top to down, at each level we descend, we add one more element into the current combination.
* The nodes marked in blue are the ones that could sum up to the target value, i.e. they are the desired combination solutions.
* The nodes marked in red are the ones that exceed the target value. Since all the candidates are positive value, there is no way we could bring the sum down to the target value, if we explore further.
* At any instant, we can **only** be at one of the nodes. When we backtrack, we are moving from a node to its parent node.

An important detail on choosing the next number for the combination is that we select the candidates **in order**, where the total candidates are treated as a list. Once a candidate is added into the current combination, we will not **look back** to all the previous candidates in the next explorations.



To demonstrate the idea, let us zoom in a node (as we shown in the above graph) to see how we can choose the next numbers.

* When we are at the node of [4], the precedent candidates are [3], and the candidates followed are [4, 5].
* We don't add the precedent numbers into the current node, since they would have been explored in the nodes in the left part of the subtree, i.e. the node of [3].
* Even though we have already the element 4 in the current combination, we are giving the element another chance in the next exploration, since the combination can contain **duplicate** numbers.
* As a result, we would branch out in two directions, by adding the element 4 and 5 respectively into the current combination.

**Algorithm**

As one can see, the above backtracking algorithm is unfolded as a DFS (Depth-First Search) tree traversal, which is often implemented with recursion.

Here we define a recursive function of backtrack(remain, comb, start) (in Python), which populates the combinations, starting from the current combination (comb), the remaining sum to fulfill (remain) and the current cursor (start) to the list of candidates. Note that, the signature of the recursive function is slightly different in Java. But the idea remains the same.

* For the first base case of the recursive function, if the remain==0, i.e. we fulfill the desired target sum, therefore we can add the current combination to the final list.
* As another base case, if remain < 0, i.e. we exceed the target value, we will cease the exploration here.
* Other than the above two base cases, we would then continue to explore the sublist of candidates as [start ... n]. For each of the candidate, we invoke the recursive function itself with updated parameters.
  + Specifically, we add the current candidate into the combination.
  + With the added candidate, we now have less sum to fulfill, i.e. remain - candidate.
  + For the next exploration, still we start from the current cursor start.
  + At the end of each exploration, we **backtrack** by popping out the candidate out of the combination.

|  |
| --- |
| class Solution {  protected void backtrack(  int remain,  LinkedList<Integer> comb,  int start, int[] candidates,  List<List<Integer>> results) {  if (remain == 0) {  // make a deep copy of the current combination  results.add(new ArrayList<Integer>(comb));  return;  } else if (remain < 0) {  // exceed the scope, stop exploration.  return;  }  for (int i = start; i < candidates.length; ++i) {  // add the number into the combination  comb.add(candidates[i]);  this.backtrack(remain - candidates[i], comb, i, candidates, results);  // backtrack, remove the number from the combination  comb.removeLast();  }  }  public List<List<Integer>> combinationSum(int[] candidates, int target) {  List<List<Integer>> results = new ArrayList<List<Integer>>();  LinkedList<Integer> comb = new LinkedList<Integer>();  this.backtrack(target, comb, 0, candidates, results);  return results;  }  } |

**Complexity Analysis**

Let N*N* be the number of candidates, T*T* be the target value, and M*M* be the minimal value among the candidates.

* Time Complexity: \mathcal{O}(N^{\frac{T}{M}+1})O(*NMT*​+1)
  + As we illustrated before, the execution of the backtracking is unfolded as a DFS traversal in a n-ary tree. The total number of steps during the backtracking would be the number of nodes in the tree.
  + At each node, it takes a constant time to process, except the leaf nodes which could take a linear time to make a copy of combination. So we can say that the time complexity is linear to the number of nodes of the execution tree.
  + Here we provide a loose upper bound on the number of nodes.
    - First of all, the fan-out of each node would be bounded to N*N*, i.e. the total number of candidates.
    - The maximal depth of the tree, would be \frac{T}{M}*MT*​, where we keep on adding the smallest element to the combination.
    - As we know, the maximal number of nodes in N-ary tree of \frac{T}{M}*MT*​ height would be N^{\frac{T}{M}+1}*NMT*​+1.
  + **Note that**, the actual number of nodes in the execution tree would be much smaller than the upper bound, since the fan-out of the nodes are decreasing level by level.
* Space Complexity: \mathcal{O}(\frac{T}{M})O(*MT*​)
  + We implement the algorithm in recursion, which consumes some additional memory in the function call stack.
  + The number of recursive calls can pile up to \frac{T}{M}*MT*​, where we keep on adding the smallest element to the combination. As a result, the space overhead of the recursion is \mathcal{O}(\frac{T}{M})O(*MT*​).
  + In addition, we keep a combination of numbers during the execution, which requires at most \mathcal{O}(\frac{T}{M})O(*MT*​) space as well.
  + To sum up, the total space complexity of the algorithm would be \mathcal{O}(\frac{T}{M})O(*MT*​).
  + Note that, we did not take into the account the space used to hold the final results for the space complexity.

#### **Similar Problems**

Once one figures out how it works with the backtracking algorithm for this problem, one can go ahead and apply this "hammer" to solve a series of similar problems.

For instance, if one goes back to the problem of [Combination Sum III](https://leetcode.com/problems/combination-sum-iii/), if suffices to tweak a bit on the invocation of the recursive function to solve the problem, in addition to some other minor adjustments on the base cases.

More specifically, in this problem, we give the current candidate another chance in the further explorations, while for the problem of [Combination Sum III](https://leetcode.com/problems/combination-sum-iii/) we simply **move on** to the candidates followed.

Here are a series of problems that one can solve, with some tweaks of the backtracking algorithm presented in this article, thanks to the great list compiled by [issac3](https://leetcode.com/problems/combination-sum/discuss/16502/A-general-approach-to-backtracking-questions-in-Java-(Subsets-Permutations-Combination-Sum-Palindrome-Partitioning)) in the discussion forum.

* [Subsets](https://leetcode.com/problems/subsets)
* [Subsets II](https://leetcode.com/problems/subsets-ii)
* [Permutations](https://leetcode.com/problems/permutations/)
* [Permutations II](https://leetcode.com/problems/permutations-ii/)
* [Combinations](https://leetcode.com/problems/combinations/)
* [Combination Sum II](https://leetcode.com/problems/combination-sum-ii/)
* [Combination Sum III](https://leetcode.com/problems/combination-sum-iii/)
* [Palindrome Partition](https://leetcode.com/problems/palindrome-partitioning/)

Approach #2: Dynamic Programming.  
subproblem dp(i, j) is defined as all the combinations of using candidates[i] (i = 0 to i) to sum to j.  
Base case: dp(0, 0) = [[]]; target 0 can be sum by no selecting any candidate  
State transition is defined as: dp(i, j) = dp[i -1, j) plus dp(i, j - candidates[i]);  
solution state: dp(n, target).  
Following solution is space optimized from 2D dp array to 1D dp array.

|  |
| --- |
| public class Solution {  public List<List<Integer>> combinationSum(int[] candidates, int target) {  List<List<Integer>>[] dp = new List[target + 1];  for (int i = 0; i <= target; i++)  dp[i] = new ArrayList<>();  dp[0].add(new ArrayList<>());  for (int candidate: candidates) {  for (int j = candidate; j <= target; j++) {  for (List<Integer> comb: dp[j - candidate]) {  List<Integer> newComb = new ArrayList(comb);  newComb.add(candidate);  dp[j].add(newComb);  }  }  }  return dp[target];  } |

**Top K Frequent Words**

Given a non-empty list of words, return the *k* most frequent elements.

Your answer should be sorted by frequency from highest to lowest. If two words have the same frequency, then the word with the lower alphabetical order comes first.

**Example 1:**

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

**Output:** ["i", "love"]

**Explanation:** "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

**Example 2:**

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

**Output:** ["the", "is", "sunny", "day"]

**Explanation:** "the", "is", "sunny" and "day" are the four most frequent words,

with the number of occurrence being 4, 3, 2 and 1 respectively.

**Note:**

1. You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
2. Input words contain only lowercase letters.

**Follow up:**

1. Try to solve it in *O*(*n* log *k*) time and *O*(*n*) extra space.

#### **Approach #1: Sorting [Accepted]**

**Intuition and Algorithm**

Count the frequency of each word, and sort the words with a custom ordering relation that uses these frequencies. Then take the best k of them.

|  |
| --- |
| 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); |

**Complexity Analysis**

* Time Complexity: O(N \log{N})*O*(*N*log*N*), where N*N* is the length of words. We count the frequency of each word in O(N)*O*(*N*) time, then we sort the given words in O(N \log{N})*O*(*N*log*N*) time.
* Space Complexity: O(N)*O*(*N*), the space used to store our candidates.

#### **Approach #2: Heap [Accepted]**

**Intuition and Algorithm**

Count the frequency of each word, then add it to heap that stores the best k candidates. Here, "best" is defined with our custom ordering relation, which puts the worst candidates at the top of the heap. At the end, we pop off the heap up to k times and reverse the result so that the best candidates are first.

In Python, we instead use heapq.heapify, which can turn a list into a heap in linear time, simplifying our work.

|  |
| --- |
| 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);  }  PriorityQueue<String> heap = new PriorityQueue<String>(  (w1, w2) -> count.get(w1).equals(count.get(w2)) ?  w2.compareTo(w1) : count.get(w1) - count.get(w2) );  for (String word: count.keySet()) {  heap.offer(word);  if (heap.size() > k) heap.poll();  }  List<String> ans = new ArrayList();  while (!heap.isEmpty()) ans.add(heap.poll());  Collections.reverse(ans);  return ans;  }  } |

**Combine Two Tables**

**Solution**

Table: Person

+-------------+---------+

| Column Name | Type |

+-------------+---------+

| PersonId | int |

| FirstName | varchar |

| LastName | varchar |

+-------------+---------+

PersonId is the primary key column for this table.

Table: Address

+-------------+---------+

| Column Name | Type |

+-------------+---------+

| AddressId | int |

| PersonId | int |

| City | varchar |

| State | varchar |

+-------------+---------+

AddressId is the primary key column for this table.

 Write a SQL query for a report that provides the following information for each person in the Person table, regardless if there is an address for each of those people:

FirstName, LastName, City, State

#### **Approach: Using outer join [Accepted]**

**Algorithm**

Since the PersonId in table **Address** is the foreign key of table **Person**, we can join this two table to get the address information of a person.

Considering there might not be an address information for every person, we should use outer join instead of the default inner join.

**MySQL**

select FirstName, LastName, City, State

from Person left join Address

on Person.PersonId = Address.PersonId

;

Note: Using where clause to filter the records will fail if there is no address information for a person because it will not display the name information.

**Rank Scores**

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

+----+-------+

| Id | Score |

+----+-------+

| 1 | 3.50 |

| 2 | 3.65 |

| 3 | 4.00 |

| 4 | 3.85 |

| 5 | 4.00 |

| 6 | 3.65 |

+----+-------+

For example, given the above Scores table, your query should generate the following report (order by highest score):

+-------+---------+

| score | Rank |

+-------+---------+

| 4.00 | 1 |

| 4.00 | 1 |

| 3.85 | 2 |

| 3.65 | 3 |

| 3.65 | 3 |

| 3.50 | 4 |

+-------+---------+

**Important Note:** For MySQL solutions, to escape reserved words used as column names, you can use an apostrophe before and after the keyword. For example**`Rank`**.