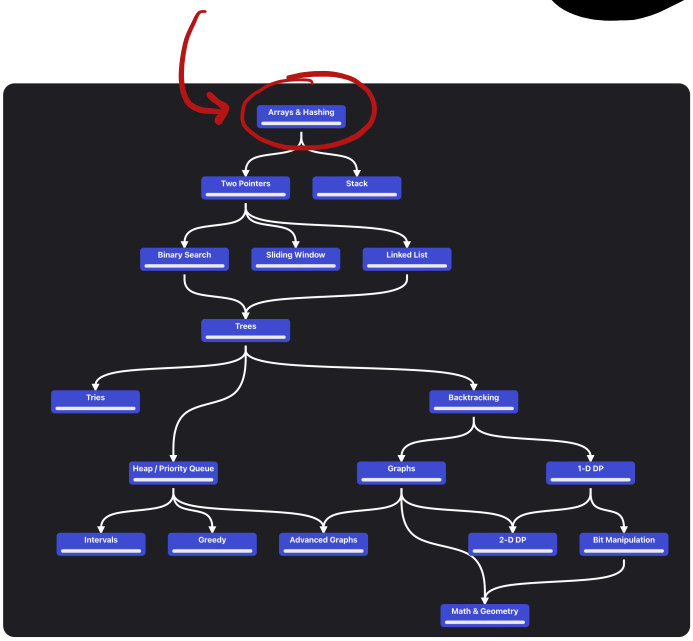
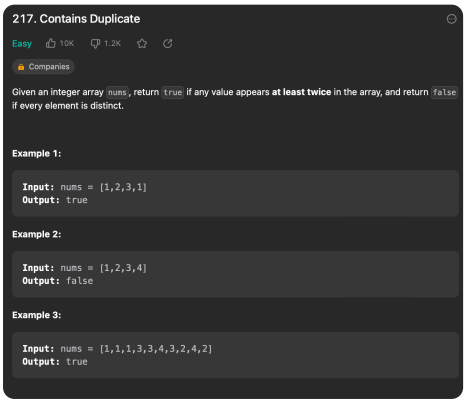
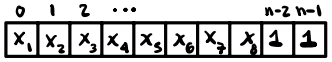
# Arrays and Hashing






# 217. Contains Duplicate




Worst Case Scenario

● = My solution

revise

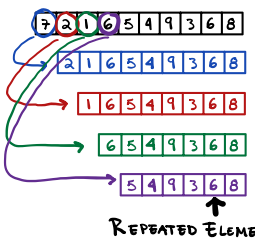
I don't think it is O(n)

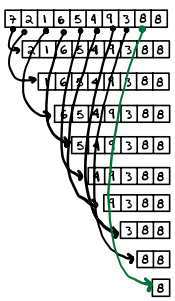
## Approach 1

Time Complexity: $O(n^2)$

Space Complexity: $O(1)$

* For each element, traverse the rest of the array to look for a repetition.



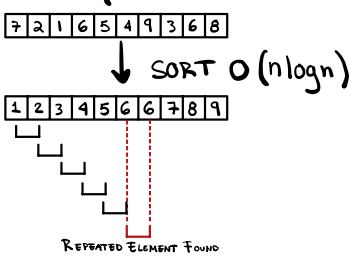
Worst Case Scenario: $O(n^2)$

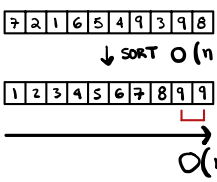
## Approach 2

Time Complexity: $O(n \log n)$

Space Complexity: $O(1)$

* Sort array first. Then look for consecutively repeated elements




Worst Case Scenario: $O(n)$

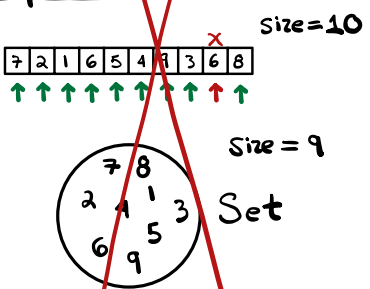
$O(n)$

## Approach 3

Time Complexity: $O(n)$

Space Complexity: $O(n)$

* Use a set to keep unrepeated elements




If both sizes are the same: NO REPEATED ELEMENTS

Else: REPEATED ELEMENTS

```cpp
bool containsDuplicate1(vector<int>& nums) {

    long n = nums.size();
    long limit = n - 1;

    for (int i = 0; i < limit; i++) {

        int e = nums[i];

        for (int j = i + 1; j < n; j++) {
            if (nums[j] == e) {
                return true;
            }
        }
    }
    return false;
}
```

```cpp
bool containsDuplicate2(vector<int>& nums) {

    sort(nums.begin(), nums.end());

    long limit = nums.size() - 1;

    for (int i = 0; i < limit; i++) {
        if (nums.at(i) == nums.at(i + 1)) {
            return true;
        }
    }

    return false;
}
```

```cpp
bool containsDuplicate3(vector<int>& nums) {

    set<int>* s = new set<int>;

    for (auto e : nums) {
        s->insert(e);
    }

    return s->size() != nums.size();
}
```

# 242. Valid Anagram

**242. Valid Anagram**

Easy 9.8K 315

Companies

Given two strings `s` and `t`, return `true` *if* `t` *is an anagram of* `s`, *and* `false` *otherwise.*

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

```
Input: s = "anagram", t = "nagaram"
Output: true
```

**Example 2:**

```
Input: s = "rat", t = "car"
Output: false
```

**Constraints:**

- `1 <= s.length, t.length <= 5 * 10^4`
- `s` and `t` consist of lowercase English letters.

**Follow up:** What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

* Count the letters' frequency of both strings, and store such frequency in arrays of size 26 whose indexes correspond to each letter in the alphabet, in order.
Then compare both arrays, element by element. (`s[i] == t[i]`)
If two elements are found to be different, return false.
Return true otherwise.

```
bool isAnagram1(string s, string t) { // Author: Ricardo Gonzalez

    const int a_ASCII = 97;
    const int alpha_len = 26;

    long s_len = s.size();
    long t_len = t.size();

    int s_letter_occur[26] = {0};
    int t_letter_occur[26] = {0};

    for (int i = 0; i < s_len; i ++) {
        s_letter_occur[s.at(i) - a_ASCII]++;
    }

    for (int i = 0; i < t_len; i ++) {
        t_letter_occur[t.at(i) - a_ASCII]++;
    }

    for (int i = 0; i < alpha_len; i ++) {
        if (s_letter_occur[i] != t_letter_occur[i]) {
            return false;
        }
    }

    return true;
}
```

→ O(n)

Compare arrays O(1)

ASCII

| | | | |
|---|---|---|---|
| 97 | 'a' | 0 | 1 |
| 98 | 'b' | 1 | 0 |
| 99 | 'c' | 2 | 2 |
| 100 | 'd' | 3 | 0 |
| 101 | 'e' | 4 | 1 |
| 102 | 'f' | 5 | 0 |
| 103 | 'g' | 6 | 0 |
| 104 | 'h' | 7 | 0 |
| 105 | 'i' | 8 | 0 |
| 106 | 'j' | 9 | 0 |
| 107 | 'k' | 10 | 0 |
| 108 | 'l' | 11 | 0 |
| 109 | 'm' | 12 | 0 |
| 110 | 'n' | 13 | 0 |
| 111 | 'o' | 14 | 0 |
| 112 | 'p' | 15 | 1 |
| 113 | 'q' | 16 | 0 |
| 114 | 'r' | 17 | 0 |
| 115 | 's' | 18 | 0 |
| 116 | 't' | 19 | 1 |
| 117 | 'u' | 20 | 0 |
| 118 | 'v' | 21 | 0 |
| 119 | 'w' | 22 | 0 |
| 120 | 'x' | 25 | 0 |
| 121 | 'y' | 24 | 0 |
| 122 | 'z' | 25 | 0 |

string s = "accept"

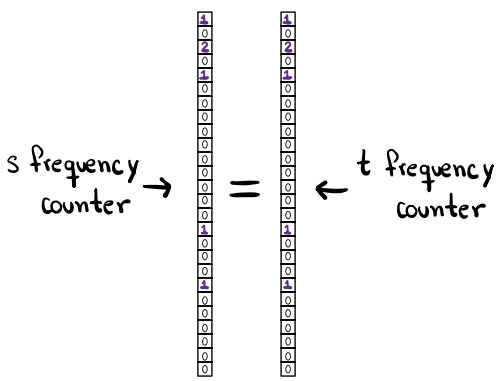
0 1 2 3 4 5 i

97 - 97 = 0 99 - 97 = 2

```
s_letter_occur[s.at(i) - a_ASCII]++;
```

char 97 index

string t = "patcec"




Time Complexity: O(n) → The strings are traversed entirely ALWAYS

Space Complexity: O(1) → The frequency counter arrays' size is CONSTANT

# Other approaches:

Sort strings. If strings are equal, valid anagram found.

string s = "classic"; → sort → s = "accilss"
string t = "siclcas"; → t = "accilss"
TRUE

string s = "rat"; → sort → s = "art"
string t = "car"; → t = "acr"
FALSE

```
bool isAnagram2(string s, string t) {

    sort(s.begin(), s.end());   → O(n log n)
    sort(t.begin(), t.end());
    return s == t;              → O(n)
}                               O(n log n) > O(n)
```

Time Complexity: O(n log n)
Space Complexity: O(1)

---

* Count the letters' frequency of string s using a single array.

Decrease the frequency of letters of string t.

Traverse the frequency array.

If an element not equal to zero is found, either 's' or 't' have more or less occurrences of the corresponding letter at that index.

s = "professor"  t = "proffessor"

After traversing s | After traversing t

| ASCII | | | count |
|---|---|---|---|
| 97 | 'a' | 0 | 0 |
| 98 | 'b' | 1 | 0 |
| 99 | 'c' | 2 | 0 |
| 100 | 'd' | 3 | 0 |
| 101 | 'e' | 4 | 1 |
| 102 | 'f' | 5 | 1 |
| 103 | 'g' | 6 | 0 |
| 104 | 'h' | 7 | 0 |
| 105 | 'i' | 8 | 0 |
| 106 | 'j' | 9 | 0 |
| 107 | 'k' | 10 | 0 |
| 108 | 'l' | 11 | 0 |
| 109 | 'm' | 12 | 0 |
| 110 | 'n' | 13 | 0 |
| 111 | 'o' | 14 | 2 |
| 112 | 'p' | 15 | 1 |
| 113 | 'q' | 16 | 0 |
| 114 | 'r' | 17 | 2 |
| 115 | 's' | 18 | 2 |
| 116 | 't' | 19 | 0 |
| 117 | 'u' | 20 | 0 |
| 118 | 'v' | 21 | 0 |
| 119 | 'w' | 22 | 0 |
| 120 | 'x' | 23 | 0 |
| 121 | 'y' | 24 | 0 |
| 122 | 'z' | 25 | 0 |

| ASCII | | | count |
|---|---|---|---|
| 97 | 'a' | 0 | 0 |
| 98 | 'b' | 1 | 0 |
| 99 | 'c' | 2 | 0 |
| 100 | 'd' | 3 | 0 |
| 101 | 'e' | 4 | 0 |
| 102 | 'f' | 5 | -1 |
| 103 | 'g' | 6 | 0 |
| 104 | 'h' | 7 | 0 |
| 105 | 'i' | 8 | 0 |
| 106 | 'j' | 9 | 0 |
| 107 | 'k' | 10 | 0 |
| 108 | 'l' | 11 | 0 |
| 109 | 'm' | 12 | 0 |
| 110 | 'n' | 13 | 0 |
| 111 | 'o' | 14 | 0 |
| 112 | 'p' | 15 | 0 |
| 113 | 'q' | 16 | 0 |
| 114 | 'r' | 17 | 0 |
| 115 | 's' | 18 | 0 |
| 116 | 't' | 19 | 0 |
| 117 | 'u' | 20 | 0 |
| 118 | 'v' | 21 | 0 |
| 119 | 'w' | 22 | 0 |
| 120 | 'x' | 23 | 0 |
| 121 | 'y' | 24 | 0 |
| 122 | 'z' | 25 | 0 |

```
bool isAnagram3(string s, string t) {
    int count[26] = {0};

    // Count the frequency of characters in string s
    for (char x : s) {
        count[x - 'a']++;
    }

    // Decrement the frequency of characters in string t
    for (char x : t) {
        count[x - 'a']--;
    }

    // Check if any character has non-zero frequency
    for (int val : count) {
        if (val != 0) {
            return false;
        }
    }

    return true;
}
```

Time Complexity: O(n)
Space Complexity: O(1)

---

Same approach.

*Note: an Unordered Map uses a hash table to store its contents.

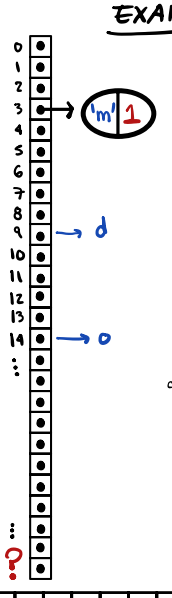
```
bool isAnagram4(string s, string t) {
    unordered_map<char, int> count;

    // Count the frequency of characters in string s
    for (auto x : s) {
        count[x]++;
    }

    // Decrement the frequency of characters in string t
    for (auto x : t) {
        count[x]--;
    }

    // Check if any character has non-zero frequency
    for (auto x : count) {          → Traversal in not in order
        if (x.second != 0) {
            return false;
        }
    }

    return true;
}
```

count[key] = value

Unordered Map (K, V)
- Node * array

```
+ K operator [] (key: K) {
        int index = hashFunction(key)
        Node cur = array[index]
        while (cur != null) {
            if (cur->key == key) {
                return cur->value
            }
        }
        return ?
}
- hashFunction(key: K): int
```

EXAMPLE  s = "mood"




count['m'] = array[3] = 'm' 1
count['d'] = array[9] = 'd' 1
count['o'] = array[14] = 'o' 2

There might be collisions, in which case:

count['o'] = array[14] = 'o' 2 → 'n' 5

# 1. Two Sum

**1. Two Sum** — Hint

Easy 49.2K 1.6K

Companies

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to* `target`.

You may assume that each input would have ***exactly one solution***, and you may not use the *same* element twice.

You can return the answer in any order.

**Example 1:**

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

**Example 2:**

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

**Example 3:**

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

O(n)

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int, int> numMap;

    for (int i = 0; i < nums.size(); ++i) {
        int complement = target - nums[i];

        // Check if the complement value is already in the map
        if (numMap.find(complement) != numMap.end()) {
            // If found, return the indices of the two numbers
            return {numMap[complement], i};
        }

        // If the complement value is not in the map, add the current value and its index to the map
        numMap[nums[i]] = i;
    }

    // If no solution is found, return an empty vector or handle the case accordingly
    return {};
}
```

O(1)

O(1)

0 1 2 3

| 1 | 7 | 11 | 2 |
|---|---|---|---|

target = 13

We traverse the vector

Iteration:

Current map state:

0

complement = 13 − 1 = 12

is complement already in map?

NO

add nums[0] to map.

1

complement = 13 − 7 = 6

is complement already in map?

NO

add nums[1] to map.

2

complement = 13 − 11 = 2

is complement already in map?

NO

add nums[2] to map.

3

complement = 13 − 2 = 11

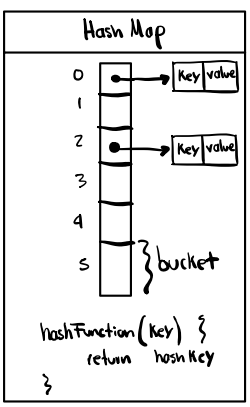
is complement already in map?

YES!

Note: 2 is not added.

## Notes:

In C++, unordered_map is implemented using a HashMap.




- HashKeys refer to **BUCKETS**

We can check one element at a time. But we can compute the other element we need (the complement) by:

nums[i] + complement = target

⇒ complement = target − nums[i]

Then we just see if such needed element is already in the map.

We map each element to its corresponding index in the vector.

**Question:**

Does a HashMap store the **key** along with the value?

Or is the key only used as a means to access the value?

Key → HashFunction → HashKey (index)

**ANSWER:** Both keys and values are stored.

# 49. Group Anagrams

## 49. Group Anagrams

Medium 16.3K 459

Companies

Given an array of strings `strs`, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

```
Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]
```

**Example 2:**

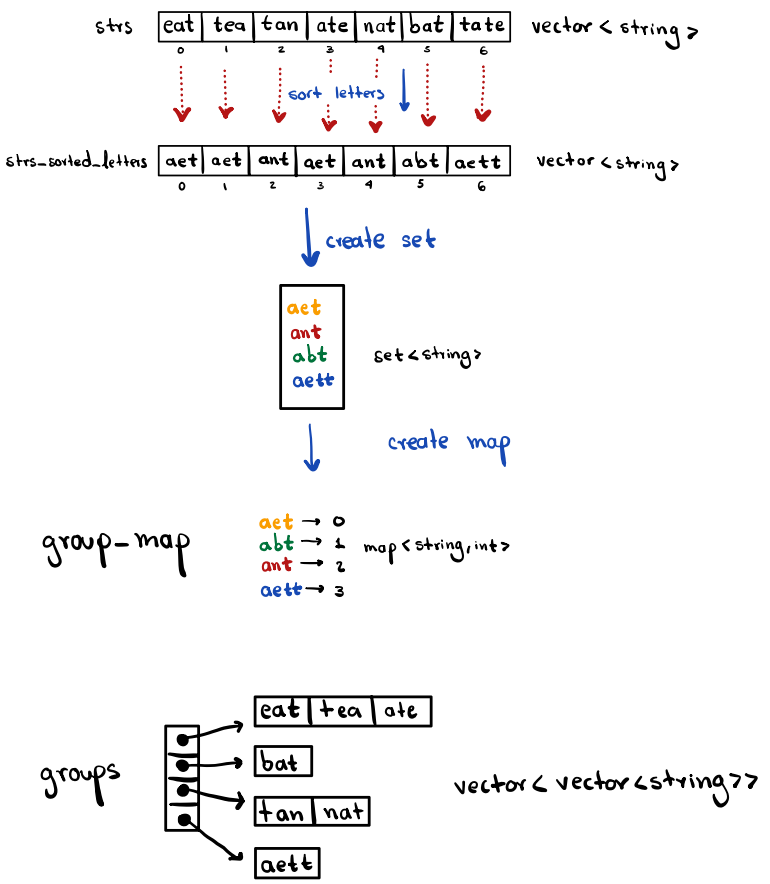
```
Input: strs = [""]
Output: [[""]]
```

**Example 3:**

```
Input: strs = ["a"]
Output: [["a"]]
```

My solution

```cpp
vector<vector<string>> groupAnagrams(vector<string>& strs) {

    unordered_set<string> temp_set; // Set to store unique sorted strings
    unordered_map<string, int> group_map; // Map to associate each sorted string with a group index

    vector<string> strs_sorted_letters(strs); // Copy the input array for sorting individual strings

    // Sort each string in the array and insert into the set to ensure uniqueness
    for (auto &el : strs_sorted_letters) {
        sort(el.begin(), el.end());
        temp_set.insert(el);
    }

    // Assign an index to each unique sorted string in the set
    int i = 0;
    for (auto &el : temp_set) {
        group_map[el] = i;
        i++;
    }

    vector<vector<string>> groups(temp_set.size()); // Vector of vectors to store groups of anagrams

    // Iterate over the original list of strings
    long l = strs.size();
    for(int i = 0; i < l; i++) {

        // Get the next word from the original array and its sorted version
        string next_word = strs[i];
        string sorted_word = strs_sorted_letters[i];

        // Add the original word to the corresponding group based on the sorted version
        groups[group_map[sorted_word]].push_back(next_word);
    }

    // Return the grouped anagrams
    return groups;
}
```



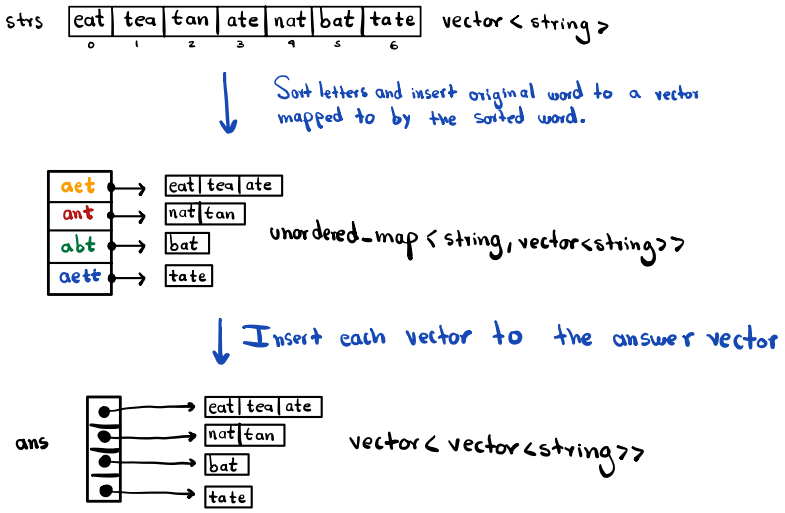
## Optimized solution

```cpp
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    // Create a hash map where each key is a sorted string, and the value is a vector of strings
    // which are anagrams of each other
    unordered_map<string, vector<string>> mp;

    for(auto x: strs) {
        string word = x;
        sort(word.begin(), word.end());
        mp[word].push_back(x);  // Add the original string 'x' to the map under the sorted key 'word'
    }

    vector<vector<string>> ans;

    for (auto x: mp) {
        // Add each group of anagrams (which is the value of the map) to the result vector
        ans.push_back(x.second);
    }

    return ans;
}
```

O(n)

**Note:** There is no need for a set, as it is created naturally.

Each element of the set maps to the collection of anagrams.



# 347. Top K Frequent Elements

Medium 15.3K 539

Companies

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:**

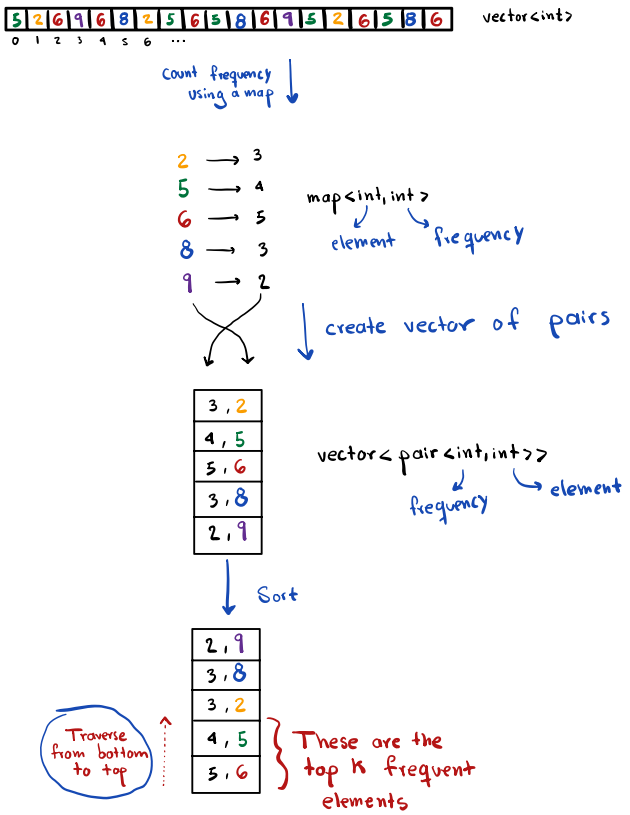
```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

**Example 2:**

```
Input: nums = [1], k = 1
Output: [1]
```

# 347. Top K Frequent Elements

Ex



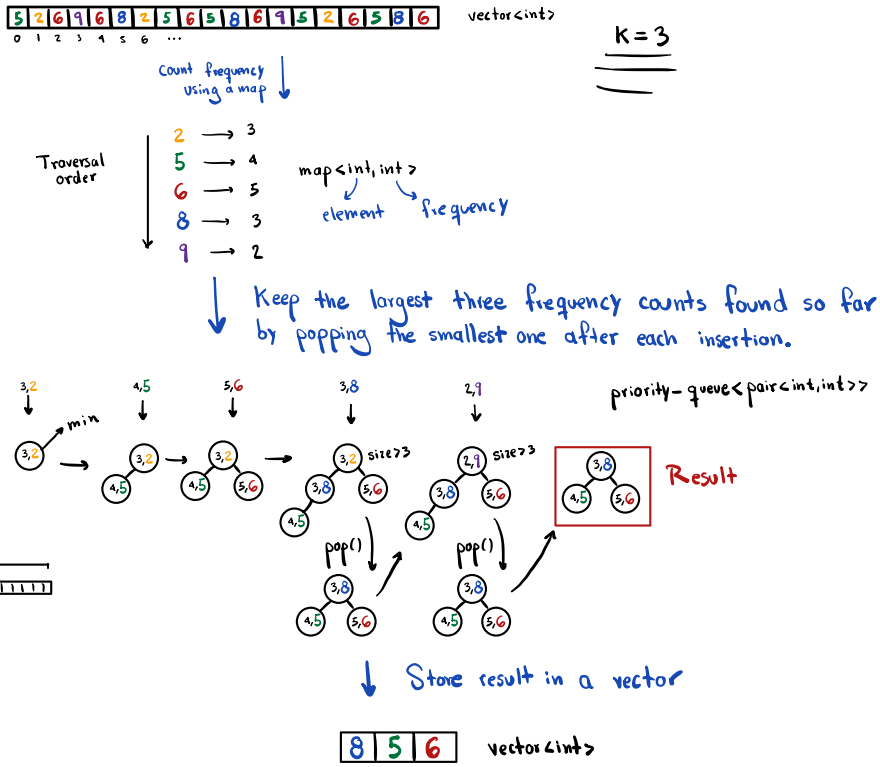

My solution

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

    unordered_map<int, int> freq_map;  //Hash map to store frequency of each element

    // Iterate over the array and update frequency count for each element
    for(auto &el: nums) {
        freq_map[el]++;
    }

    // Create a vector to store pairs of frequency and element
    vector<pair<int,int>> freq_el_pair;
    // Iterate over the frequency map
    for(auto &el: freq_map) {
        // Insert a pair of frequency (first) and element (second) into the vector
        freq_el_pair.push_back(pair(el.second, el.first));
    }

    // Sort the vector of pairs. The pairs are sorted first by frequency
    sort(freq_el_pair.begin(), freq_el_pair.end());

    vector<int> ans;
    long l = freq_el_pair.size();
    // Add the top k elements to the answer vector
    for(int i = 0; i < k; i++) {
        // The elements are added in reverse order since the highest frequencies are at the end
        ans.push_back(freq_el_pair[l - i - 1].second);
    }

    // Return the answer vector containing the top k frequent elements
    return ans;
}
```

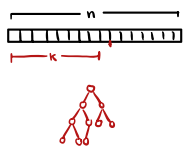
# Using a Min-Heap

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

    unordered_map<int, int> freq_map; // Hash map to store frequency of each element

    // Iterate over the array and update frequency count for each element
    for (int num : nums) {
        freq_map[num]++;
    }

    // Create a min-heap (priority queue) to store the k most frequent elements
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> min_heap;

    // Iterate over the frequency map
    for (auto& it : freq_map) {
        // Add the element and its frequency as a pair to the min-heap
        min_heap.push({it.second, it.first});
        // If the size of the min-heap exceeds k, pop the least frequent element
        if (min_heap.size() > k) {
            min_heap.pop();
        }
    }

    vector<int> top_k;
    // Extract the elements from the min-heap
    while (!min_heap.empty()) {
        top_k.push_back(min_heap.top().second);
        min_heap.pop();
    }

    return top_k;
}
```

Ex




Notes: The time complexity of insertion in a heap is $O(\log k)$.
Then inserting $n$ elements takes $O(n \log k)$.
As $k \to n$, $O(n \log k) \to O(n \log n)$



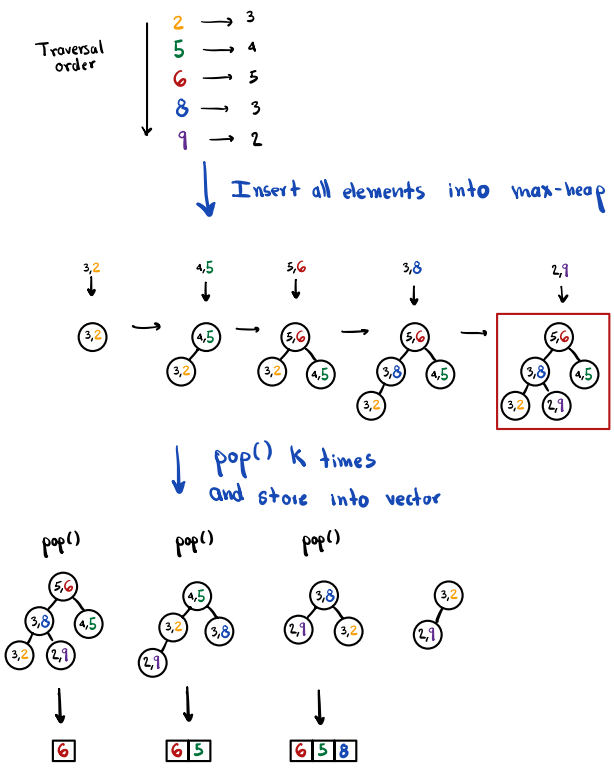

---

# Using a Max-Heap

```cpp
vector<int> topKFrequent(vector<int>& nums, int k) {
    // Create a hash map to store frequency of each element
    unordered_map<int, int> freq_map;
    for (int num : nums) {
        freq_map[num]++; O(1) } O(n)
    }

    // Create a max-heap (priority queue) to store elements by frequency
    priority_queue<pair<int, int>> max_heap;

    // Iterate over the frequency map
    for (auto& it : freq_map) {
        // Add the element and its frequency as a pair to the max-heap
        max_heap.push({it.second, it.first}); m = # of unique elements
    }

    // Prepare the answer vector
    vector<int> top_k;
    // Extract the top k elements from the max-heap
    while (k-- > 0 && !max_heap.empty()) {
        top_k.push_back(max_heap.top().second);
        max_heap.pop();
    }

    // Return the answer vector containing the top k frequent elements
    return top_k;
}
```

O(m log m)

O(m)




Note: This approach is less memory-efficient as we are storing all unique elements in the max-heap.

# 36. Valid Sudoku

**36. Valid Sudoku**

Medium 9.2K 961

Companies

Determine if a `9 x 9` Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits `1-9` without repetition.
2. Each column must contain the digits `1-9` without repetition.
3. Each of the nine `3 x 3` sub-boxes of the grid must contain the digits `1-9` without repetition.

**Note:**

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

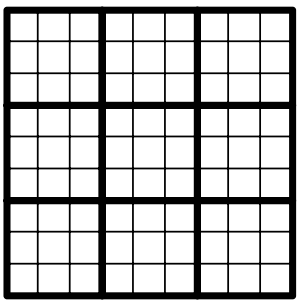
**Example 1:**

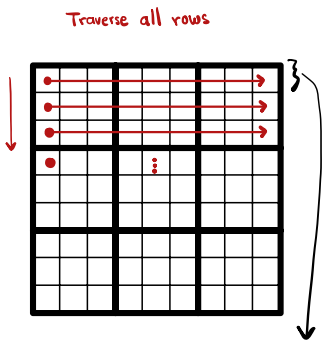
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 5 | 3 | | | 7 | | | | |
| 6 | | | 1 | 9 | 5 | | | |
| | 9 | 8 | | | | | 6 | |
| 8 | | | | 6 | | | | 3 |
| 4 | | | 8 | | 3 | | | 1 |
| 7 | | | | 2 | | | | 6 |
| | 6 | | | | | 2 | 8 | |
| | | | 4 | 1 | 9 | | | 5 |
| | | | | 8 | | | 7 | 9 |

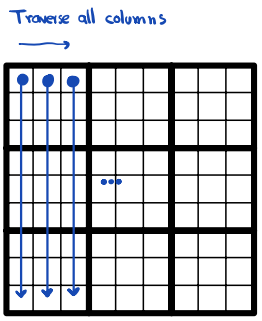
**Input:** board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
**Output:** true

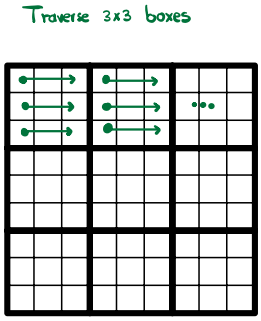
**Example 2:**

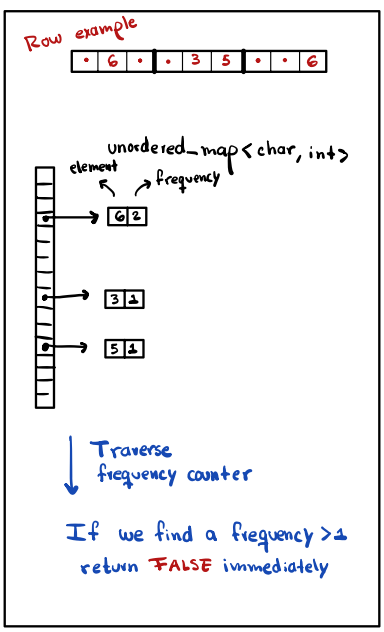
**Input:** board =
[["8","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
**Output:** false
**Explanation:** Same as Example 1, except with the **5** in the top left corner being modified to **8**. Since there are two 8's in the top left 3x3 sub-box, it is invalid.






SAME

SAME

```cpp
bool isValidSudoku(vector<vector<char>>& board) {

    // Validate rows
    long num_rows = board.size();
    long num_cols = board[0].size();


    for (int i = 0; i < num_rows; i++) {
        if (!validateRow(board, num_cols, i)) {
            return false;
        }
    }

    // Validate columns
    for (int j = 0; j < num_cols; j++) {
        if (!validateCol(board, num_rows, j)) {
            return false;
        }
    }

    // Validate boxes
    for (int box_row = 0; box_row < 3; box_row++) {
        for (int col_row = 0; col_row < 3; col_row++) {
            if (!validateBox(board, 3 * box_row, 3 * col_row)) {
                return false;
            }
        }
    }

    return true;
}
```

```cpp
bool validateRow(vector<vector<char>>& board, long row_size, int row_num) {
    unordered_map<char, int> freq_counter;

    for (int j = 0; j < row_size; j++ ) {
        if ( board[row_num][j] != '.') { // First check if the character at i, col_num is NOT '.'
            freq_counter [ board[row_num][j] ]++;
        }
    }

    for (auto &el: freq_counter) {
        if (el.second > 1) {
            return false;
        }
    }
    return true;
}

// Return true if col is valid
bool validateCol(vector<vector<char>>& board, long col_size, int col_num) {
    unordered_map<char, int> freq_counter;

    for (int i = 0; i < col_size; i++ ) {
        if ( board[i][col_num] != '.') { // First check if the character at i, col_num is NOT '.'
            freq_counter[ board[i][col_num] ]++;
        }
    }

    for (auto &el: freq_counter) {
        if (el.second > 1) {
            return false;
        }
    }
    return true;
}

bool validateBox(vector<vector<char>> &board, int start_row, int start_col) {
    unordered_map<char, int> freq_counter;

    for (int i = start_row; i < start_row + 3; i++) {

        for (int j = start_col; j < start_col + 3; j++) {
            if ( board[i][j] != '.') { // First check if the character at i, col_num is NOT '.'
                freq_counter[ board[i][j] ]++;
            }
        }
    }

    for (auto &el: freq_counter) {
        if (el.second > 1) {
            return false;
        }
    }

    return true;
}
```

chatGPT

```cpp
bool isValidSudoku(vector<vector<char>>& board) {
    // Use fixed-size arrays of 9 elements for each row, column, and box
    array<array<int, 9>, 9> rows = {0}, cols = {0}, boxes = {0};

    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            char c = board[i][j];
            if (c != '.') {
                int index = c - '1'; // Convert char to int index (0-8)
                int boxIndex = (i / 3) * 3 + j / 3; // Calculate the box index

                // Check for duplicates and increment the count
                if (++rows[i][index] > 1 || ++cols[j][index] > 1 || ++boxes[boxIndex][index] > 1) {
                    return false;
                }
            }
        }
    }

    return true;
}
```

---

LeetCode

```cpp
bool isValidSudoku(std::vector<std::vector<char>>& board) {
    // Using maps to keep track of all numbers in each column, row, and square
    std::unordered_map<int, std::unordered_set<char>> cols;
    std::unordered_map<int, std::unordered_set<char>> rows;
    std::unordered_map<int, std::unordered_set<char>> squares;

    // Iterating over each cell in the 9x9 Sudoku board
    for (int r = 0; r < 9; ++r) {
        for (int c = 0; c < 9; ++c) {
            // Skip the iteration for empty cells
            if (board[r][c] == '.') {
                continue;
            }

            // Store the current number
            char num = board[r][c];

            // Calculate the index of the 3x3 square this cell belongs to
            int boxIndex = (r / 3) * 3 + c / 3;

            // Check if the current number is already present in the current row, column, or square
            if (rows[r].count(num) || cols[c].count(num) || squares[boxIndex].count(num)) {
                return false; // If number is present, the board is invalid
            }

            // If the number is not present, add it to the current row, column, and square set
            cols[c].insert(num);
            rows[r].insert(num);
            squares[boxIndex].insert(num);
        }
    }

    // If no duplicates are found, the board is valid
    return true;
}
```

# 128. Longest Consecutive Sequence

Medium 17.3K 749

Companies

Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in `O(n)` time.

Thus we can't sort the array

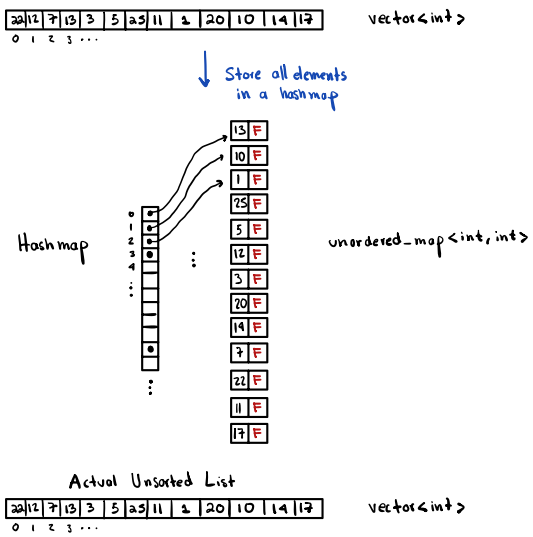
**Example 1:**

```
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.
```

**Example 2:**

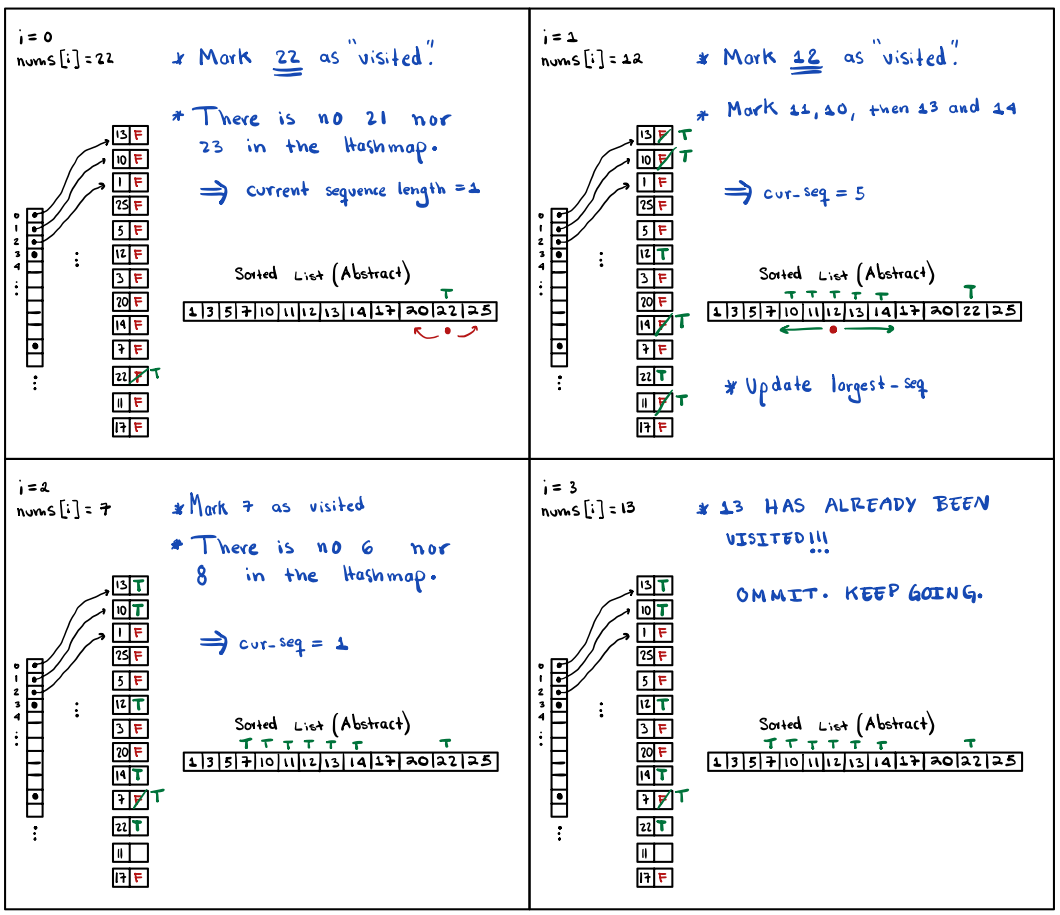
```
Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9
```

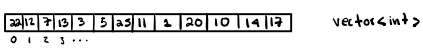
# 128. Longest Consecutive Sequence




```cpp
int longestConsecutive(vector<int>& nums) {

    unordered_map<int, bool> nums_visited;

    long l = nums.size();

    for (int i = 0; i < l; i++) {
        int next_num = nums[i];
        nums_visited[next_num] = false;
    }


    int largest_seq = 0;

    for (int i = 0; i < l; i++) {

        int cur_num = nums[i];
        bool visited = nums_visited[cur_num];
        int cur_seq = 1;

        if (!visited) {

            nums_visited[cur_num] = true;
            int n = cur_num;

            while (nums_visited.find(n - 1) != nums_visited.end()) {
                nums_visited[n - 1] = true;
                cur_seq++;
                n = n - 1;
            }

            n = cur_num;

            while (nums_visited.find(n + 1) != nums_visited.end()) {
                nums_visited[n + 1] = true;
                cur_seq++;
                n = n + 1;
            }

            if (largest_seq < cur_seq) {
                largest_seq = cur_seq;
            }
        }
    }
    return largest_seq;
}
```

**Note:** If an element is already visited, its sequence it belongs to has already been found.




ts

IS NOT in map

IS in map

## Chat GPT

```cpp
int longestConsecutive(vector<int>& nums) {
    unordered_map<int, int> seqLength;  // Maps a number to the length of its sequence
    int largest_seq = 0;

    for (int num : nums) {
        if (seqLength.find(num) == seqLength.end()) { NOT FOUND
            int left = seqLength.find(num - 1) != seqLength.end() ? seqLength[num - 1] : 0;
            int right = seqLength.find(num + 1) != seqLength.end() ? seqLength[num + 1] : 0;

            // Sum up the length of the sequence
            int sum = left + right + 1;
            seqLength[num] = sum;

            // Update the max length for each number
            largest_seq = max(largest_seq, sum);

            // Extend the length to the boundary of the sequence
            // This avoids checking every element in the sequence
            seqLength[num - left] = sum;
            seqLength[num + right] = sum;
        }
    }
    return largest_seq;
}
```

## Actual Unsorted List

vector<int>

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | ... |
|---|---|---|---|---|---|---|---|---|
| 11 | 10 | 12 | 14 | 15 | 13 | 16 | 3 | 25 | 20 | 7 | 5 | 1 |

### i = 0, nums[i] = 11

20 IS NOT in map ⇒ l = 0
12 IS NOT in map ⇒ r = 0
Sum = l + r + 1 = 0

Update sequence length in current and boundaries:
- map[num[i]] = sum = 2
- map[num[i] − 0] = sum = 2
- map[num[i] + 0] = sum = 2

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | 1 | | | | | | | |

### i = 1, nums[i] = 10

9 IS NOT in map ⇒ l = 0
11 IS in map ⇒ r = 1
Sum = l + r + 1 = 2

Update sequence length in current and boundaries:
- map[num[i]] = sum = 2
- map[num[i] − 0] = sum = 2
- map[num[i] + 1] = sum = 2

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | 2 | 2 | | | | | | | |

### i = 2, nums[i] = 12

11 IS in map ⇒ l = 2
13 IS NOT in map ⇒ r = 0
Sum = l + r + 1 = 3

Update sequence length in current and boundaries:
- map[num[i]] = sum = 3
- map[num[i] − 2] = sum = 3
- map[num[i] + 0] = sum = 3

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | | 3 | | | | | | |

### i = 3, nums[i] = 14

13 IS NOT in map ⇒ l = 0
15 IS NOT in map ⇒ r = 0
Sum = l + r + 1 = 1

Update sequence length in current and boundaries:
- map[num[i]] = sum = 1
- map[num[i] − 0] = sum = 1
- map[num[i] + 0] = sum = 1

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | 1 | | | | |

### i = 4, nums[i] = 15

14 IS in map ⇒ l = 1
16 IS NOT in map ⇒ r = 0
Sum = l + r + 1 = 2

Update sequence length in current and boundaries:
- map[num[i]] = sum = 2
- map[num[i] − 1] = sum = 2
- map[num[i] + 0] = sum = 2

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | | | | | 2 | | | |

### i = 5, nums[i] = 13

12 IS in map ⇒ l = 3
14 IS in map ⇒ r = 2
Sum = l + r + 1 = 6

Update sequence length in current and boundaries:
- map[num[i]] = sum = 6
- map[num[i] − 3] = sum = 6
- map[num[i] + 2] = sum = 6

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | | | 6 | | | | | |

### i = 6, nums[i] = 16

15 IS in map ⇒ l = 6
17 IS NOT in map ⇒ r = 0
Sum = l + r + 1 = 7

Update sequence length in current and boundaries:
- map[num[i]] = sum = 7
- map[num[i] − 6] = sum = 7
- map[num[i] + 0] = sum = 7

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | | | | | | 7 | | |

### i = {7, ..., 12}, nums[i] = 3, 25, 20, 7, 5, 1

nums[i] − 1 IS NOT in map ⇒ l = 0
nums[i] + 1 IS NOT in map ⇒ r = 0
Sum = l + r + 1 = 1

Update sequence length in current and boundaries:
- map[num[i]] = sum = 1
- map[num[i] − 0] = sum = 1
- map[num[i] + 0] = sum = 1

Sorted List (Abstract)

| 1 | 2 | 3 | 5 | 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 20 | 25 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 3 | | | | | | | | | | | |

Final values:

| 1 | 1 | 1 | 1 | 7 | 2 | 3 | 6 | 2 | 6 | 7 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

LeetCode

```cpp
int longestConsecutive(std::vector<int>& nums) {
    // Initialize an unordered set with the elements of nums.
    // This set will contain unique elements for quick access.
    unordered_set<int> numSet(nums.begin(), nums.end());

    // This will keep track of the longest consecutive sequence found.
    int longest = 0;

    // Iterate through each number in the original vector.
    for (int n : nums) {
        // Check if the current number is the start of a sequence.
        // This is true if the number just before it (n - 1) is not in the set.
        if (numSet.find(n - 1) == numSet.end()) {
            // Initialize the length of the current sequence.
            int length = 0;

            // Continue checking for the next numbers in the sequence.
            // This while loop increments the length for each consecutive number found.
            while (numSet.find(n + length) != numSet.end()) {
                length++;
            }

            // Update the longest sequence length found so far.
            // std::max will ensure we always have the max between the current and the new length.
            longest = max(length, longest);
        }
    }

    // After checking all numbers, return the length of the longest consecutive sequence.
    return longest;
}
```

# 238. Product of Array Except Self

**238. Product of Array Except Self**

Medium 18.8K 1K

Companies

Given an integer array `nums`, return *an array* `answer` *such that* `answer[i]` *is equal to the product of all the elements of* `nums` *except* `nums[i]`.

The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

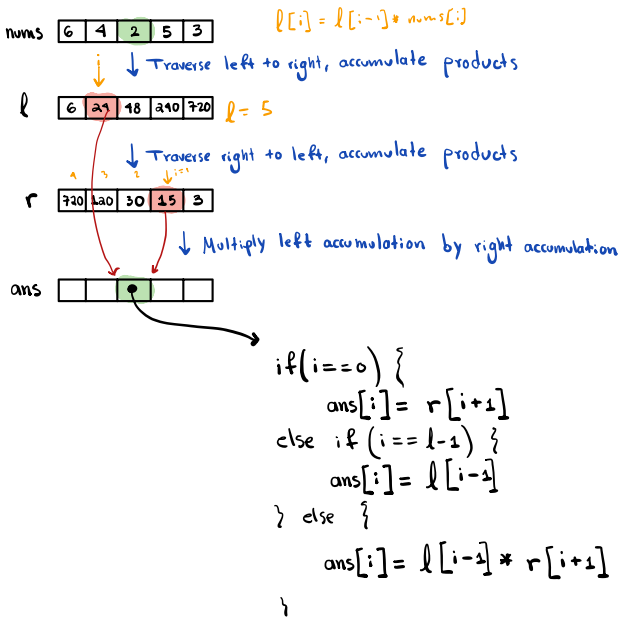
You must write an algorithm that runs in `O(n)` time and without using the division operation.

**Example 1:**

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

**Example 2:**

```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
```




```
if(i==0) {
    ans[i] = r[i+1]
else if (i==l-1) {
    ans[i] = l[i-1]
} else {
    ans[i] = l[i-1] * r[i+1]
}
```

```cpp
vector<int> productExceptSelf(vector<int>& nums) {

    long l = nums.size();

    int left_pr_acc[l];
    int right_pr_acc[l];

    left_pr_acc[0] = nums[0];
    right_pr_acc[l-1] = nums[l-1];

    for (int i = 1; i < l; i++) {
        left_pr_acc[i] = left_pr_acc[i - 1] * nums[i];
        right_pr_acc[l-1-i] = right_pr_acc[l-i] * nums[l-1-i];
    }

    vector<int> ans;
    for (int i = 0; i < l; i++) {
        if (i == 0) {
            ans.push_back(right_pr_acc[i+1]);
        } else if (i == (l - 1)) {
            ans.push_back(left_pr_acc[i-1]);
        } else {
            ans.push_back(left_pr_acc[i-1] * right_pr_acc[i+1]);
        }
    }

    return ans;
}
```

```cpp
vector<int> productExceptSelf(vector<int>& nums) {
    long n = nums.size();
    vector<int> ans(n, 1);

    // First pass: calculate left products
    for (int i = 1; i < n; i++) {
        ans[i] = ans[i - 1] * nums[i - 1];
    }

    // Variable to store right product
    int rightProd = 1;

    // Second pass: calculate right products and multiply with left products
    for (long i = n - 1; i >= 0; i--) {
        ans[i] *= rightProd;
        rightProd *= nums[i];
    }

    return ans;
}
```