Leetcode practice

Tuesday, June 4, 2024 11:01 AM

1. Two sum (Easy)

The Two Sum problem asks us to find two numbers in an array that sum up to a given target value. We need to return the indices of these two numbers.

From <https://leetcode.com/problems/two-sum/solutions/3619262/3-method-s-c-java-python-beginner-friendly/>

Brute Force

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();
        for (int i = 0; i< n; i++){
            for (int j = i + 1; j < n; j++){
                if (nums[i] + nums[j] == target){
                    return {i, j};
                }
            }
        }
        return {};
    }
};
```

2. Roman to integer

The key intuition lies in the fact that in Roman numerals, when a smaller value appears before a larger value, it represents subtraction, while when a smaller value appears after or equal to a larger value, it represents addition.

From <https://leetcode.com/problems/roman-to-integer/solutions/3651672/best-method-c-java-python-beginner-friendly/>

```
#include <unordered_map>
#include <iostream>
#include <string>
class Solution {
public:
    int romanToInt(std::string s) {
        std::unordered_map<char, int> m;
        // Define the values for each Roman numeral
        m['I'] = 1;
        m['V'] = 5;
        m['X'] = 10;
        m['L'] = 50;
        m['C'] = 100;
        m['D'] = 500;
        m['M'] = 1000;
        int ans = 0;
        // Iterate through the string
        for (int i = 0; i < s.length(); i++) {
            // Check if the current value is less than the next value
            if (i < s.length() - 1 && m[s[i]] < m[s[i + 1]]) {
                ans -= m[s[i]];
            } else {
                ans += m[s[i]];
            }
```

```
        }
        return ans;
    }
};

int main() {
    Solution solution;
    std::string roman = "MCMXCIV"; // Example Roman numeral
    int result = solution.romanToInt(roman);
    std::cout << "The integer value of " << roman << " is: " << result
<< std::endl;
    return 0;
}
```

For the example "IX":

- When i is 0, the current character s[i] is 'I'. Since there is a next character ('X'), and the value of 'I' (1) is less than the value of 'X' (10), the condition m[s[i]] < m[s[i+1]] is true. In this case, we subtract the value of the current character from ans.
  ans -= m[s[i]];
  ans -= m['I'];
  ans -= 1;
  ans becomes -1.
- When i is 1, the current character s[i] is 'X'. This is the last character in the string, so there is no next character to compare. Since there is no next character, we don't need to evaluate the condition. In this case, we add the value of the current character to ans.
  ans += m[s[i]];
  ans += m['X'];
  ans += 10;
- ans becomes -1 +10 = 9

3. Palindrome number

The intuition behind this code is to reverse the entire input number and check if the reversed number is equal to the original number. If they are the same, then the number is a palindrome

From <https://leetcode.com/problems/palindrome-number/solutions/3651712/2-method-s-c-java-python-beginner-friendly/>

```
#include <iostream>
using namespace std;
int main() {
  int n, reversed_number = 0, remainder;
  cout << "Enter an integer: ";
  cin >> n;
  while(n != 0) {
    remainder = n % 10;
    reversed_number = reversed_number * 10 + remainder;
    n /= 10;
  }
  cout << "Reversed Number = " << reversed_number;
  return 0;
}
```

n	n != 0	remainder	reversed_number
2345	true	5	0 * 10 + 5 = 5
234	true	4	5 * 10 + 4 = 54
23	true	3	54 * 10 + 3 = 543
2	true	2	543 * 10 + 2 = 5432
0	false	-	Loop terminates.

4.Maxiumum Subarray

The Intuition behind the code is to find the maximum sum of a contiguous subarray within the given array nums. It does this by scanning through the array and keeping track of the current sum of the subarray. Whenever the current sum becomes greater than the maximum sum encountered so far, it updates the maximum sum. If the current sum becomes negative, it resets the sum to 0 and starts a new subarray. By the end of the loop, the code returns the maximum sum found.

From <https://leetcode.com/problems/maximum-subarray/solutions/3666304/beats-100-c-java-python-beginner-friendly/>

```cpp
#include <iostream>
#include <vector>
#include <climits>
class Solution {
public:
    int maxSubArray(std::vector<int>& nums) {
        int maxSum = INT_MIN;
        int currentSum = 0;
        for (int i = 0; i < nums.size(); i++) {
            currentSum += nums[i];
            if (currentSum > maxSum) {
                maxSum = currentSum;
            }

            if (currentSum < 0) {
                currentSum = 0;
            }
        }
        return maxSum;
    }
};
int main() {
    Solution solution;
    std::vector<int> nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4}; // Example input
    int result = solution.maxSubArray(nums);
    std::cout << "The maximum subarray sum is " << result << std::endl;
```

```
    return 0;
}
```

Array = [-2, -3, 4, -1, -2, 1, 5, -3]

Index	Array								Curr_Sum	Max_Sum
i=0	-2	-3	4	-1	-2	1	5	-3	-2	-2
i=1	-2	-3	4	-1	-2	1	5	-3	-3	-2
i=2	-2	-3	4	-1	-2	1	5	-3	4	4
i=3	-2	-3	4	-1	-2	1	5	-3	3	4
i=4	-2	-3	4	-1	-2	1	5	-3	1	4
i=5	-2	-3	4	-1	-2	1	5	-3	2	4
i=6	-2	-3	4	-1	-2	1	5	-3	7	7
i=7	-2	-3	4	-1	-2	1	5	-3	4	7

5.Remove Element

The intuition behind this solution is to iterate through the array and keep track of two pointers: index and i. The index pointer represents the position where the next non-target element should be placed, while the i pointer iterates through the array elements. By overwriting the target elements with non-target elements, the solution effectively removes all occurrences of the target value from the array.

From <https://leetcode.com/problems/remove-element/solutions/3670940/best-100-c-java-python-beginner-friendly/>

```
#include <iostream>
#include <vector>
#include <climits>
class Solution {
public:
    int removeElement(std::vector<int>& nums, int val) {
        // Represents the current position for the next non-target element
        int index = 0;
        for (int i = 0; i < nums.size(); i++) {
            // If nums[i] is not equal to val, it means it is a non-target
element
            if (nums[i] != val) {
                nums[index] = nums[i];
                index++;
            }
        }
        return index;
    }
```

```
};
int main() {
    Solution solution;
    std::vector<int> nums = {0, 1, 2, 2, 3, 0, 4, 2}; // Initialize the
vector correctly
    int val = 2;
    int newLength = solution.removeElement(nums, val);

    std::cout << "The array after removing the element " << val << " is: ";
    for (int i = 0; i < newLength; i++) {
        std::cout << nums[i] << " ";
    }
    std::cout << std::endl;
    std::cout << "The new length of the array is " << newLength << std::endl;

    return 0;
}
```

6.contain duplicate

The hash set approach uses a hash set data structure to store encountered elements. It iterates through the array, checking if an element is already in the set. If so, it returns true. Otherwise, it adds the element to the set. This approach has a time complexity of O(n) and provides an efficient way to check for duplicates.

From <https://leetcode.com/problems/contains-duplicate/solutions/3672475/4-method-s-c-java-python-beginner-friendly/>

```
#include <iostream>
#include <vector>
#include <unordered_set>

class Solution {
public:
    bool containsDuplicate(std::vector<int>& nums) {
        std::unordered_set<int> check_duplicate;
        for (int num : nums) {
            //std::unordered_set::find() function returns an iterator
pointing to  the element if it is found in the set,
            //and it returns std::unordered_set::end() if the element
is not found
            if (check_duplicate.find(num) != check_duplicate.end()) {
                return true;
            }
            check_duplicate.insert(num);
        }
        return false;
    }
};

int main() {
    Solution solution;
    std::vector<int> nums = {1, 2, 3, 4, 5, 6, 7, 8, 9, 1}; // Example
input
    bool result = solution.containsDuplicate(nums);
    if (result) {
```

```
        std::cout << "The array contains duplicates." << std::endl;
    } else {
        std::cout << "The array does not contain duplicates."
<< std::endl;
    }
    return 0;
}
```

7.Add two Numbers

The Intuition is to iterate through two linked lists representing non-negative integers in reverse order, starting from the least significant digit. It performs digit-wise addition along with a carry value and constructs a new linked list to represent the sum. The process continues until both input lists and the carry value are exhausted. The resulting linked list represents the sum of the input numbers in the correct order.

From <https://leetcode.com/problems/add-two-numbers/solutions/3675747/beats-100-c-java-python-beginner-friendly/>

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */

ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    // Dummy head node to simplify edge cases and result list
construction
    ListNode* dummyHead = new ListNode(0);
    ListNode* p = l1; // Pointer to traverse the first list
    ListNode* q = l2; // Pointer to traverse the second list
    ListNode* curr = dummyHead; // Pointer to construct the result
list
    int carry = 0; // Initialize carry to 0
    // Loop through both linked lists
    while (p != nullptr || q != nullptr) {
        // Extract values from the current nodes of both lists
        int x = (p != nullptr) ? p->val : 0;
        int y = (q != nullptr) ? q->val : 0;
        // Calculate the sum of the two digits and the carry
        int sum = carry + x + y;
        // Update the carry for the next position
        carry = sum / 10;

        int digit = sum%10;

        // Create a new node with the digit value (sum % 10)
        curr->next = new ListNode(digit);
        // Move to the next node in the result list
```

```
            curr = curr->next;
            // Advance the pointers p and q if they are not null
            if (p != nullptr) p = p->next;
            if (q != nullptr) q = q->next;
        }
        // If there is a remaining carry, add a new node with the carry
    value
        if (carry > 0) {
            curr->next = new ListNode(carry);
        }
        // The result list starts from the next node of dummyHead
        ListNode* head = dummyHead->next;
        // Delete the dummy head node to avoid memory leak
        delete dummyHead;
        return head;
    }
```

1. **Initialization**:
   - dummyHead is created.
   - p points to the head of l1, q points to the head of l2.
   - carry is 0.
2. **First Iteration**:
   - x = 2, y = 5, sum = 2 + 5 + 0 = 7.
   - carry = 0, new node with value 7 is created and appended to the result list.
   - Move p and q to the next nodes.
3. **Second Iteration**:
   - x = 4, y = 6, sum = 4 + 6 + 0 = 10.
   - carry = 1, new node with value 0 is created and appended to the result list.
   - Move p and q to the next nodes.
4. **Third Iteration**:
   - x = 3, y = 4, sum = 3 + 4 + 1 = 8.
   - carry = 0, new node with value 8 is created and appended to the result list.
   - Move p and q to the next nodes.
5. **Final Carry Check**:
   - No remaining carry, so no additional node is added.

The final result list is 7 -> 0 -> 8, which represents the number 807 (342 + 465 = 807).

8.Majority Element

The intuition behind using a hash map is to count the occurrences of each element in the array and then identify the element that occurs more than n/2 times. By storing the counts in a hash map, we can efficiently keep track of the occurrences of each element.

From <https://leetcode.com/problems/majority-element/solutions/3676530/3-methods-beats-100-c-java-python-beginner-friendly/>

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        unordered_map<int, int> m;  // To store the frequency of each
    element
        int n = nums.size();        // Size of the array
```

```
        // Count the frequency of each element
        for (int i = 0; i < nums.size(); i++) {
            m[nums[i]]++;
        }
        n = n / 2;  // Calculate the threshold for majority element
        // Iterate through the map to find the majority element
        for (auto x : m) {
            if (x.second > n) {
                return x.first;  // Return the element if its count is
greater than n/2
            }
        }
        return 0;  // Return 0 if no majority element found (not
required as per problem constraints)
    }
};
```

Consider the array nums = [2, 2, 1, 1, 1, 2, 2].

1. **Count Frequency**:
   - m[2]++: Now m is {2: 1}
   - m[2]++: Now m is {2: 2}
   - m[1]++: Now m is {2: 2, 1: 1}
   - m[1]++: Now m is {2: 2, 1: 2}
   - m[1]++: Now m is {2: 2, 1: 3}
   - m[2]++: Now m is {2: 3, 1: 3}
   - m[2]++: Now m is {2: 4, 1: 3}
2. **Calculate Threshold**:
   - n = nums.size() / 2: n = 7 / 2 = 3
3. **Find Majority Element**:
   - Iterate through the map m:
     - For element 2: m[2] = 4 which is greater than 3. Hence, 2 is the majority element.

The function returns 2.

9.Remove duplicated from sorted array

The Intuition is to use two pointers, i and j, to iterate through the array. The variable j is used to keep track of the current index where a unique element should be placed. The initial value of j is 1 since the first element in the array is always unique and doesn't need to be changed.

From <https://leetcode.com/problems/remove-duplicates-from-sorted-array/solutions/3676877/best-method-100-c-java-python-beginner-friendly/>

```
#include <iostream>
#include <vector>
class Solution {
public:
    int removeDuplicates(std::vector<int>& nums) {
        if (nums.empty()) return 0; // Handle edge case where the
input array is empty
        int j = 1;
        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] != nums[i - 1]) {
```

```
                nums[j] = nums[i];
                j++;
            }
        }
        return j;
    }
};
int main() {
    Solution solution;
    std::vector<int> nums = {1, 2, 2, 2, 5, 6, 7, 8, 9, 1}; // Example
input
    int newLength = solution.removeDuplicates(nums);

    std::cout << "The array after removing duplicates is: ";
    for (int i = 0; i < newLength; i++) {
        std::cout << nums[i] << " ";
    }
    std::cout << std::endl;

    std::cout << "The new length of the array is: " << newLength
<< std::endl;
    return 0;
}
```

10.Valid anagram

The Intuition is to determine if two strings are anagrams, compare the characters in both strings and check if they have the same characters but in a different order. By tracking the count of each character, if the counts match for all characters, the strings are anagrams; otherwise, they are not.

From <https://leetcode.com/problems/valid-anagram/solutions/3687854/3-methods-c-java-python-beginner-friendly/>

```
#include <iostream>
#include <unordered_map>
#include <string>
using namespace std;
class Solution {
public:
    bool isAnagram(string s, string t) {
        unordered_map<char, int> count;

        // Count the frequency of characters in string s
        for (auto x : s) {
            count[x]++;
        }
        // Decrease the frequency of characters based on string t
        for (auto x : t) {
            count[x]--;
        }
        // Check if all counts are zero
        for (auto x : count) {
            if (x.second != 0) {
```

```
                return false;
            }
        }
        return true;
    }
};
int main() {
    Solution solution;
    string s = "listen;
    string t = "silent";
    if (solution.isAnagram(s, t)) {
        cout << "The strings are anagrams." << endl;
    } else {
        cout << "The strings are not anagrams." << endl;
    }
    return 0;
}
```

11.group anagrams

The intuition is to group words that are anagrams of each other together. Anagrams are words that have the same characters but in a different order.

From <https://leetcode.com/problems/group-anagrams/solutions/3687735/beats-100-c-java-python-beginner-friendly/>

- x.first is the key of the map entry (in this case, a sorted string).
- x.second is the value of the map entry (in this case, a std::vector<std::string> containing all the original strings that are anagrams of each other).
- **With &**: More efficient because it avoids copying. Only references to the elements are used.

```
#include <iostream>
#include <unordered_map>
#include <string>
#include <vector>
#include <algorithm>  // Needed for sort()
using namespace std;
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> mp;
        // Iterate over each string in the input list
        for (auto x : strs) {
            string word = x;  // Copy the current string
            sort(word.begin(), word.end());  // Sort the string to get
the anagram key
            mp[word].push_back(x);  // Use the sorted string as the
key and append the original string to the map
        }
        vector<vector<string>> ans;  // Prepare to collect the result
        for (auto x : mp) {
            ans.push_back(x.second);  // Append each group of anagrams
to the result
```

```
        }
        return ans;
    }
};
int main() {
    vector<string> strs = {"eat", "tea", "tan", "ate", "nat", "bat"};
    Solution solution;
    vector<vector<string>> result = solution.groupAnagrams(strs);
    cout << "The group anagrams are: " << endl;
    for (const auto& group : result) {
        cout << "[ ";
        for (const auto& word : group) {
            cout << word << " ";
        }
        cout << "]" << endl;
    }
    return 0;
}
```

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> charmp;
        int left = 0;
        int maxLength = 0;

        for (int right = 0; right < s.length(); right++) {
            if (charmp.count(s[right]) == 0 || charmp[s[right]]
< left) {
                charmp[s[right]] = right;
                maxLength = max(maxLength, right - left + 1);
            } else {
                left = charmp[s[right]] + 1;
                charmp[s[right]] = right;
            }
        }
        return maxLength;
    }
};
```

12. Longest Substring Without Repeating Characters

The intuition behind the 3 solutions is to iteratively find the longest substring without repeating characters by maintaining a sliding window approach. We use two pointers (left and right) to represent the boundaries of the current substring. As we iterate through the string, we update the pointers and adjust the window to accommodate new unique characters and eliminate repeating characters.

From <https://leetcode.com/problems/longest-substring-without-repeating-characters/solutions/3649636/3-method-s-c-java-python-beginner-friendly/>

- **First Character: 'a' (Index 0)**
  - right = 0
  - 'a' is not in charmp.
  - Update charmp['a'] = 0.
  - Calculate maxLength = max(0, 0 - 0 + 1) = 1.
  - State: charmp = {'a': 0}, maxLength = 1, left = 0
- **Second Character: 'b' (Index 1)**
  - right = 1
  - 'b' is not in charmp.
  - Update charmp['b'] = 1.
  - Calculate maxLength = max(1, 1 - 0 + 1) = 2.
  - State: charmp = {'a': 0, 'b': 1}, maxLength = 2, left = 0
- **Third Character: 'c' (Index 2)**
  - right = 2
  - 'c' is not in charmp.
  - Update charmp['c'] = 2.
  - Calculate maxLength = max(2, 2 - 0 + 1) = 3.
  - State: charmp = {'a': 0, 'b': 1, 'c': 2}, maxLength = 3, left = 0
- **Fourth Character: 'a' (Index 3)**
  - right = 3
  - 'a' is in charmp and charmp['a'] >= left (0 >= 0).
  - Update left = charmp['a'] + 1 = 1.
  - Update charmp['a'] = 3.
  - Calculate maxLength = max(3, 3 - 1 + 1) = 3.
  - State: charmp = {'a': 3, 'b': 1, 'c': 2}, maxLength = 3, left = 1
- **Fifth Character: 'b' (Index 4)**
  - right = 4
  - 'b' is in charmp and charmp['b'] >= left (1 >= 1).
  - Update left = charmp['b'] + 1 = 2.
  - Update charmp['b'] = 4.
  - Calculate maxLength = max(3, 4 - 2 + 1) = 3.
  - State: charmp = {'a': 3, 'b': 4, 'c': 2}, maxLength = 3, left = 2