Two pointers collection

Derrick Guo

February 2020

The situations to use 2 pointers: 2 input arrays / strings; linked list cycle / mid point; input is sorted, etc (will continue to add cases here)

1 15. 3Sum

Given an array nums of n integers, are there elements a, b, c in nums such that a + b + c = 0? Find all unique triplets in the array which gives the sum of zero.

Note:

The solution set must not contain duplicate triplets.

Example:

Given array nums = [-1, 0, 1, 2, -1, -4],

A solution set is: [[-1, 0, 1], [-1, -1, 2]]

idea:

This is a very classic 2 pointer problem. First we sort the array, then we call sorted 2 sum. Since the array is sorted, we can use 2 pointers: one pointing to the beginning and one pointing to the end. If the sum of the 2 pointers > target, we move the right pointer to the left; otherwise we move the left pointer to the right. We don't find the target if the left and right pointer come across each other. We call this sorted 2 sum on all the elements in the array, and return the unique triplets.

```python
class Solution(object):
    def threeSum(self, nums):
        """
        :type nums: List[int]
        :rtype: List[List[int]]
        """
        nums.sort()
        res = []
        i = 0
        while i < len(nums):
            n = nums[i]
            left, right = i + 1, len(nums) - 1
            target = -n
            while left < right:
                cursum = nums[left] + nums[right]
                if cursum == target:
                    res.append([n, nums[left], nums[right]])
                    while left < right and nums[left] == res[-1][1]:
                        left += 1
                    while left < right and nums[right] == res[-1][2]:
                        right -= 1
                elif cursum > target:
                    right -= 1
                else:
                    left += 1
            while i < len(nums) - 1 and nums[i] == nums[i+1]:
                i += 1
            i += 1
        return res
```

2 16. 3Sum Closest

Given an array nums of n integers and an integer target, find three integers in nums such that the sum is closest to target. Return the sum of the three integers. You may assume that each input would have exactly one solution.

Example:

Given array nums = [-1, 2, 1, -4], and target = 1.

The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

idea:

This question is very similar to 3 sum. The only thing that we need to keep in mind is to track the current smallest difference between our cursum and target, and the result we want to return. We update them when we compute 3 sum, then we're done!

```
class Solution(object):
    def threeSumClosest(self, nums, target):
        """
        :type nums: List[int]
        :type target: int
        :rtype: int
        """
        nums.sort()
        res = float('inf')
        closest_diff = float('inf')
        i = 0
        while i < len(nums):
            left, right = i + 1, len(nums) - 1
            newtarget = target - nums[i]
            while left < right:
                cursum = nums[left] + nums[right]
                if cursum == newtarget:
                    return target
                if abs(cursum - newtarget) < closest_diff:
                    closest_diff = abs(cursum - newtarget)
                    res = cursum + nums[i]
                if cursum < newtarget:
                    left += 1
                else:
                    right -= 1
            i += 1
        return res
```

3 316. Remove Duplicate Letters

Given a string which contains only lowercase letters, remove duplicate letters so that every letter appears once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example 1:

Input: ”bcabc” Output: ”abc” Example 2:

Input: ”cbacdcbc” Output: ”acdb”

idea:

Since we need the result to be the smallest in lexicographical order among all possible results, we need a counter to count how many each character appears in the input string, so that we know, when we find a smaller character, if we want to remove it or not. For example, ”bcabc”. When we have ”bc” in our result, we meet ”a”. The reason we know we can remove ”b” and ”c” is that we have a counter dictionary, and until this point we have more ”b” and ”c” in front of us that we haven't met. So we can remove ”b” and ”c” and put ”a” into the result.

We also need a visited dictionary to indicate if the current character is in the result. For example, if we have ”a” in the result, and when we meet another ”a”, we just ignore it.

This is not a classic 2 pointer problem, but it does require us to point one pointer to the input string and another pointer to the result string. If we found a char from the input string that is smaller than the last char of the result string, and there is more of the last char of the result string coming, we can get rid of it in the result string.

```python
class Solution(object):
    def removeDuplicateLetters(self, s):
        """
        :type s: str
        :rtype: str
        """
        counter = collections.Counter(s)
        visited = collections.defaultdict(bool)

        res = ''
        for c in s:
            counter[c] -= 1
            if visited[c]: continue
            while res and res[-1] > c and counter[res[-1]] > 0:
                visited[res[-1]] = False
                res = res[:-1]
            res += c
            visited[c] = True
        return res
```

4 977. Squares of a Sorted Array

Given an array of integers A sorted in non-decreasing order, return an array of the squares of each number, also in sorted non-decreasing order.

Example 1:

Input: [-4,-1,0,3,10] Output: [0,1,9,16,100] Example 2:

Input: [-7,-3,2,3,11] Output: [4,9,9,49,121]

idea:

The normal idea is to sort the array by absolute value first and compute the squares. That would be O(nlogn). We can do it faster since the array is sorted already. Since we know the lenght of the input array, thus we know the length of the output array. Then we can instantiate result array with all element 0 of length n, then fill it using two pointers. One pointer pointing to the left and one to the right. If abs(nums[left]) $>$ abs(nums[Right]) then we fill the res with nums[left] $**$ 2 and move left to the right by 1. Otherwise we fill the res array with nums[right] $**$ 2 and move right to the left by 1. We keep doing this until right = left, at that time we filled the entire result array. This is O(n).

```python
class Solution(object):
    def sortedSquares(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
        res = [0 for i in range(len(A))]
        left, right = 0, len(A) - 1
        res_ptr = len(A)-1
        while left <= right:
            if abs(A[left]) > abs(A[right]):
                res[res_ptr] = A[left] ** 2
                left += 1
            else:
                res[res_ptr] = A[right] ** 2
                right -= 1
            res_ptr -= 1
        return res
```

5 3. Longest Substring Without Repeating Characters

Given a string, find the length of the longest substring without repeating characters.

Example 1:

Input: "abcabcbb" Output: 3 Explanation: The answer is "abc", with the length of 3. Example 2:

Input: "bbbbb" Output: 1 Explanation: The answer is "b", with the length of 1. Example 3:

Input: "pwwkew" Output: 3 Explanation: The answer is "wke", with the length of 3. Note that the answer must be a substring, "pwke" is a subsequence and not a substring.

idea:

Each time we see a duplicate char that is currently in our substring, we want to include the new one and get rid of all chars before the old one. For example, if the input is "fgabcabc", and we are looking at "fgabc", then when we see the next char which is "a", we want to include this new "a", and throw away everything before the old "a", which gives us "bca", and then we expand from there. Thus we use a dictionary to track the last seen index of a character. If we have seen a char before, and it is in the substring we are tracking, then we need to exclude everything before (and including) this seen char and add the latest instance into our substring. Otherwise we update the cur length and the max length.

```
class Solution(object):
    def sortedSquares(self, A):
        """
        :type A: List[int]
        :rtype: List[int]
        """
```

```python
        res = [0 for i in range(len(A))]
        left, right = 0, len(A) - 1
        res_ptr = len(A)-1
        while left <= right:
            if abs(A[left]) > abs(A[right]):
                res[res_ptr] = A[left] ** 2
                left += 1
            else:
                res[res_ptr] = A[right] ** 2
                right -= 1
            res_ptr -= 1
        return res
```

6 142. Linked List cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

To represent a cycle in the given linked list, we use an integer pos which represents the position (0-indexed) in the linked list where tail connects to. If pos is -1, then there is no cycle in the linked list.

Note: Do not modify the linked list.

Example 1:

Input: head = [3,2,0,-4], pos = 1 Output: tail connects to node index 1 Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:

Input: head = [1,2], pos = 0 Output: tail connects to node index 0 Explanation: There is a cycle in the linked list, where tail connects to the first node.

Example 3:

Input: head = [1], pos = -1 Output: no cycle Explanation: There is no cycle in the linked list.

idea:
Fast slow pointer.
Let the distance from head to cycle entry point be d, fast and slow meets at a position where it is x unit distance from entry point, r be the length of the cycle. Since fast covers twice distance as slow, we know that $2(d+x) = d+x+n*r$. Then we get $d+x = n*r$. Since we want to find the entry point, we want to find d. Then since $d = n*r-x$, we let one pointer starts from the head, and the other one starts from where fast and slow meet, then when head goes d, the other pointer goes $n*r-x$ which is exactly the entry point since the distance from entry point to meeting point is x.

```python
class Solution:
    # @param head, a ListNode
    # @return a list node
    def detectCycle(self, head):
        try:
            fast = head.next
            slow = head
            while fast is not slow:
                fast = fast.next.next
                slow = slow.next
        except:
            # if there is an exception, we reach the end and there is no cycle
            return None

        # since fast starts at head.next, we need to move slow one step forward
        slow = slow.next
        while head is not slow:
            head = head.next
            slow = slow.next

        return head
```

7 876. Middle of the Linked List

Given a non-empty, singly linked list with head node head, return a middle node of linked list.

If there are two middle nodes, return the second middle node.

Example 1:

Input: [1,2,3,4,5] Output: Node 3 from this list (Serialization: [3,4,5]) The returned node has value 3. (The judge's serialization of this node is [3,4,5]). Note that we returned a ListNode object ans, such that: ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, and ans.next.next.next = NULL. Example 2:

Input: [1,2,3,4,5,6] Output: Node 4 from this list (Serialization: [4,5,6]) Since the list has two middle nodes with values 3 and 4, we return the second one.

idea:

Fast and slow both starts from the head, fast goes twice as fast as slow, and when fast reaches the end, which means fast is null or fast.next is null, then slow reaches the middle of the list.

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None
```

```

class Solution(object):
    def middleNode(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head: return None
        slow = fast = head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        return slow
```

8 202. Happy Number

Write an algorithm to determine if a number is ”happy”.

A happy number is a number defined by the following process: Starting with any positive integer, replace the number by the sum of the squares of its digits, and repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1. Those numbers for which this process ends in 1 are happy numbers.

Example:

Input: 19 Output: true Explanation: 12 + 92 = 82 82 + 22 = 68 62 + 82 = 100 12 + 02 + 02 = 1

idea:

The easier way to solve this is to use a set to keep track of all the numbers we encountered. If we ever come across a number that is in the seen set before we get to 1, then we know that we are in a cycle and the number is not happy. This uses extra space.

If we want constant space, we can use fast & slow pointer. fast goes twice as fast as slow, and if fast catches slow and they don’t get to 1, it means the number is not happy. The loop breaks when one of them get to 1, which means the number is happy. Both solutions are below.

```
class Solution(object):
    def isHappy(self, n):
        """
        :type n: int
        :rtype: bool
        """
        if n == 1: return True
        visited = set()

        while n not in visited:
```

```python
            visited.add(n)
            cur_sum = 0
            for c in str(n):
                cur_sum += int(c) ** 2
            if cur_sum == 1:
                return True
            n = cur_sum
        return False
```

```python
class Solution(object):
    def isHappy(self, n):
        """
        :type n: int
        :rtype: bool
        """
        if n == 1: return True
        slow = fast = n

        def step(num):
            s = str(num)
            res = 0
            for c in s:
                res += int(c)**2
            return res

        while slow != 1 and fast != 1:
            slow = step(slow)
            fast = step(step(fast))
            if slow == fast and slow != 1:
                return False
        return True
```