Contents

leetcode: curated75

Numbering as per list
Leetcode: Blind Curated 75 https://leetcode.com/list/xoqag3yj/

(1) Two Sum [Array]

- hash map (dict) used for index lookup of diffrence (target-firstNumber)

```python
def twosum1(nums, target):
    lookup_dict = {n: i for i, n in enumerate(nums)}
    for i, num in enumerate(nums):
        difference = target - num
        if difference in lookup_dict:
            if i != lookup_dict.get(difference):  # looked up yourself?
                return [i, lookup_dict.get(difference)]
```

(2) Longest Substring Without Repeating Characters [String]

- sliding window with left, right pointer (indeces) of current longest substr

```python
def longest_substring(s):
    if not len(s):
        return 0
    left = 0
    Sright = 0

    curr_substr = set(s[0])
    max_length = 1

    for i in range(1, len(s)):
        right = i
        right_char = s[right]
        if right_char not in curr_substr:
            curr_substr.add(right_char)
            max_length = max(max_length, len(curr_substr))
        else:
            while c := s[left]:  # delete chars left of found duplicate
                curr_substr.remove(c)
                left += 1
                if c == right_char:
                    break
            curr_substr.add(right_char)
    return max_length
```

(3) Longest Palindromic Substring [String]

```python
    # ideal / better solution
    result = ""
    result_length = 0
    for i in range(len(s)):
        # odd
        left, right = i, i  # start with just center char
```

```
    while (left >= 0 and right < len(s)) and (s[left] == s[right]):
        current_length = right - left + 1
        if current_length > result_length:
            result_length = current_length
            result = s[left:right+1]
        # move outwards:
        left  -= 1
        right += 1
    # even
    left, right = i, i+1
    while left >= 0 and right < len(s) and (s[left] == s[right]):
        current_length = right - left + 1
        if current_length > result_length:
            result_length = current_length
            result = s[left:right+1]
        # move outwards:
        left  -= 1
        right += 1
return result


# own solution (messy)

# one length string is a palindrome
# two length string is a palindrome if both letters same
#  odd string from center..yes if all outwards pairs same letter
#  even center is 2 letters and need be same, from then on same as for odd

str_length = len(s)
if str_length == 0:
    return ""
if str_length == 1 or (str_length == 2 and s[0] == s[1]):
    return s
if str_length == 2:
    return s[0]

longest = s[0]  # if all differenct chars min is one char e.g. 'abcdef'

# wrong: length at least three start with third char - would miss aba
for i in range(1, str_length):
    center = s[i]  # odd center (one char)
    if s[i-1] == s[i]:  # even center (two chars)
        center_even = s[i-1] + s[i]
        if len(center_even) > len(longest):
            longest = center_even
    else:
        center_even = None

    # odd:
    # s[i-1]  s   s[i+1]  if good this is new center
    # s[i-2]  s   s[i+2]  ...and continue
    max_range = min(i, str_length-i-1)
    for j in range(1, max_range+1):
        # if neither off nor even is palindrome
        # at this stage...skip rest
        if not center and not center_even:
            break

        left  = s[i-j]
        right = s[i+j] # same for odd and even
        left_for_even = "" # one further out left
        if i-1-j >= 0:
            left_for_even = s[i-1-j]

        # is new palindrome for odd?
        if center and left == right:
            # string concat in each loop: rather expensive!
            # better/faster: save l/r index in tuple or even 2 ints
            center = left + center + right
        else: # break i.e. don't continue checking outwards
            center = None

        # is new palindrome for even?
        if center_even and left_for_even == right:
            center_even = left_for_even + center_even + right
        else: # break i.e. don't continue checking outwards
            center_even = None

        if center and len(center) > len(longest):
            longest = center
        if center_even and len(center_even) > len(longest):
            longest = center_even
return longest
```

(5) 3Sum

- sort list! iterate through list, devide problem into current number (nums[i]) + – two sum with left/right pointer

```
def threeSum(self, nums: List[int]) -> List[List[int]]:
    results = []
    nums.sort()  # !!!!
    target = 0
    for i  in range(0, len(nums)-1):
        if i > 0 and nums[i] == nums[i-1]:
            continue
        if nums[i] > target:  # lowest already > target in sorted list
            break
        left, right = i+1, len(nums)-1
        while left < right:
            s = nums[i] + nums[left] + nums[right]
            if s == target:
                results.append([nums[i], nums[left], nums[right]])
                # update 'left' below

            if s <= target:
                left = left + 1
                while nums[left] == nums[left-1] and left < right:
                    left = left + 1
            elif s > target:
                right = right - 1
    return results
```

(6) Remove Nth Node From End of List [Linked List]

Given the `head` of a linked list, remove the nth node from the end of the list and return its head.

```
def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
    dummy = ListNode(-1, head) # dummy.next = head
    left = dummy # so we arrive at n-1 to remove n
    right = head
    # move right ahead by n
    while n > 0:
        right = right.next
        n -= 1
    # move both pointer by 1 each until end
    # (pointer distance remains at n)
    while right:
        left = left.next
        right = right.next
    # remove node after left
    left.next = left.next.next
    return dummy.next
```

(7) Valid Parenthesis [String]

Given a string `s` containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

```
# ...one version
matching = {")":"(", "]":"[", "}":"{"}
if len(s) == 1 or s[0] not in matching.values():
    return False
opening = deque()
for p in s:
    if p in matching.values(): # dict values are opening ones
        opening.appendleft(p)
    else: # closing one
        corresponding_opening = matching.get(p)
        if opening:  # could be empty if closing before opening
            opening_from_stack = opening.popleft()
        else:
            opening_from_stack = None
        if not corresponding_opening == opening_from_stack:
            return False
if opening: # if not empty then not all open ones where closed
    return False
return True

# second version
parentheses = {'(': ')', '{': '}', '[': ']'}
next_expected_closing = []
for i, p in enumerate(s):
    if p in parentheses.keys():  # # opening parentheses
        next_expected_closing.append(parentheses[p])
    else:  # closing...
        if not next_expected_closing:  # parantheses was not opened
            return False
        # validate it's the correct closing type
        closing = next_expected_closing.pop()
        if not p == closing:
```

```
            return False
    # no pending closing parenthesis
    if not next_expected_closing:
        return True
    else:
        return False
```

(8) Merge Two Sorted Lists [Linked List]

Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

```
# def mergeTwoLists1(self, l1: ListNode, l2: ListNode) -> ListNode:
    new_start_node = ListNode()
    current_end = new_start_node
    while l1 and l2:
        if l1.val < l2.val:
            current_end.next = l1
            l1 = l1.next
        else:
            current_end.next = l2
            l2 = l2.next
        current_end = current_end.next
    if l1:
        current_end.next = l1
    elif l2:
        current_end.next = l2

    return new_start_node.next

# def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
    if not l1:
        return l2
    elif not l2:
        return l1

    node_l1 = l1
    node_l2 = l2
    prev = ListNode(-1000)
    head = prev
    while True:
        if node_l1.val <= node_l2.val:
            prev.next = node_l1
            prev = node_l1
            node_l1 = node_l1.next
        elif node_l1.val > node_l2.val:
            prev.next = node_l2
            prev = node_l2
            node_l2 = node_l2.next

        if not node_l1:
            prev.next = node_l2
            break
        if not node_l2:
            prev.next = node_l1
            break

    return head.next
```

(13) Group Anagrams

```
def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
    if not strs or len(strs) == 0:
        return []
    anagrams = defaultdict(list)
    for word in strs:
        key = ''.join(sorted(word))
        anagrams[key].append(word)

    return [arr for arr in anagrams.values()]
```

(14) Maximum Subarray [Array, DP]

- sliding window (sort of), dynamic programming

```
if len(nums) == 0:
    return 0

max_subarr_sum = nums[0]
curr_subarr_sum = nums[0]
#  for right in range(1, len(nums)):
for number in nums[1:]:
    if number > (curr_subarr_sum + number):
        curr_subarr_sum = number
    else:
        curr_subarr_sum += number
    max_subarr_sum = max(max_subarr_sum, curr_subarr_sum)
```

```
return max_subarr_sum
```

(16) Jump Game [Dynamic Programming]

Given an array of non-negative integers nums, you are initially positioned at the first index of the array.Each element in the array represents your maximum jump length at that position. Determine if you are able to reach the last index.

```
# e.g. [2,3,1,1,4]
# 0 -> 0 + 2  maxreach = index 2
# 1 -> 1 + 3  maxreach = index 4
# 2 -> 2 + 1             index 3 ...max(maxreachsofar, 3)
if not len or len(nums) == 1:
    return True
max_reach_so_far = 0
for i, n in enumerate(nums):
    if i > max_reach_so_far:  # outrun!
        break
    reach = i + n
    max_reach_so_far = max(reach, max_reach_so_far)
return max_reach_so_far >= len(nums)-1
```

(20) Climbing Stairs [Dynamic Programming]

You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Framework for Solving DP Problems:

1. Define the objective function
2. Identify base cases
3. Write down a recurrence relation for the optimized obj. func e.g `f(n) = f(n-1) + f(n-2)`
4. What's the order of execution? e.g.bottom-up
5. Where to look for the answer? e.g. f(n)

```
if n == 1: return 1
if n == 2: return 2
dp: List = [None]  * (n + 1)
dp[0] = 1  # also just one distinct way to get here
dp[1] = 1
dp[2] = 2  #  = dp[1] + dp[0]
dp[3] = 3  #  = dp[2] + dp[1]
for i in range(4, n+1):
    dp[i] = dp[i-1] + dp[i-2]
return dp[n]
```

(26) Same Tree [Tree]

Given the roots of two binary trees p and q, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

```
def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
    if not p and not q:  # base/edge cases...
        return True
    if not p or not q:  # one of both is empty
        return False    # already know (above) not both empty
    if p.val != q.val:  # validated with above both not empty
        return False
    # now we know both not empty and root value is same
    # continue recursively comparing nodes
    return (self.isSameTree(p.left, q.left) and
            self.isSameTree(p.right, q.right))
```

(28) Maximum Depth of Binary Tree [Tree]

Given the `root` of a binary tree, return its maximum depth. A binary tree's maximum depth is the nr of nodes along the longest path from the root down to the farthest leaf.

```
def height(self, node: TreeNode) -> int:
    if node == None:
        return 0
    return 1 + max(self.height(node.left), self.height(node.right))
```

(30) Best Time to Buy and Sell Stock [Array, DP]

`prices[i]` is the price of a given stock on the ith day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

- Complexity:
  - Time- *O(n)* only a single pass is needed
  - Space *O(1)* two variables

```
# e.g. prices = [7, 1, 5, 3, 6, 4]  # result: 5
```

```
# Solution 1
max_profit, buy = 0, sys.maxsize

for _, sell in enumerate(prices):
    if sell < buy:
        buy = sell
    else:
        profit = sell - buy
        max_profit = max(max_profit, profit)
return(max_profit)

# Solution 2
min_price = prices[0]
max_profit = 0
for i in range(1, len(prices)):
    if prices[i] < min_price:
        min_price = prices[i]
    profit = prices[i] - min_price
    max_profit = max(profit, max_profit)
return max_profit
```

(32) Valid Palindrome [String]

```
s = "A man, a plan, a canal: Panama"
# normalize str e.g. to "amanaplanacanalpanama"
p = str()
for c in s:
    if c.isalnum():
        p += c.lower()

length = len(p)

if length in [0, 1]:
    return True

for i in range(length):
    left = i
    right = (length - 1) - i
    if not left <= right:
        return True
    if p[left] != p[right]:
        return False
```

(36) Linked List Cycle [Linked List]

Given **head**, the head of a linked list, determine if the linked list has a cycle in it.

```
def hasCycle(self, head: ListNode) -> bool:
    if not head:  # head.next == None ?
        return False
    been_here_before = set()
    curr_node = head
    while curr_node:
        if curr_node in been_here_before:
            return True
        been_here_before.add(curr_node)
        curr_node = curr_node.next
    return False
```

(40) Reverse Bits [Binary]

Reverse bits of a given 32 bits unsigned integer.

```
# n = 43261596  # result: 964176192
result = 0
number_of_bits = 32
for b in reversed(range(number_of_bits)): # reversed index 0..31
    # get rightmost bit, and reverse index position
    # bit at i=0, is new index 31 i.e. shift left by 'b' positions
    result |= (n & 1) << b
    # move orinigal number one to the right
    # i.e. just processed bit 'falls off'
    n >>= 1
return result

# alternative leveraging python bin/str functions
reversed_binary = ''.join(reversed(bin(n)[2:].zfill(32)))
return int(reversed_binary, 2)
```

(41) Number of 1 bits [Binary]

Write a function that takes an unsigned integer and returns the number of '1' bits it has (also known as the *Hamming weight*).

```
def hammingWeight(self, n: int) -> int:
    return str(bin(n)).count("1")
```

```
# counting using shifting
def bitCount(number:int) -> int:
count = 0
length = 32
for _ in range(32):
    count += (number & 1)
    number >>= 1
return count
```

(42) House Robber [Dynamic Programming]

the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night. Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

```
    # example: nums = [1,2,3,1]    output: 4
    # index=0   nums[0]
    # index=1   max( nums[0], nums[1])
    # index=2   max(r[i-2] + nums[i] or r[1])
    if len(nums) == 1: return nums[0]
    if len(nums) == 2: return max(nums[0], nums[1])

    dp = [None]*len(nums)

    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])
    dp[2] = max(dp[0]+nums[2], dp[1])

    for i in range(3, len(nums)):
        max_rob_so_far = max(dp[i-2]+nums[i], dp[i-1])
        dp[i] = max_rob_so_far

    return max(dp) # max not necessary (is in last element)

    #
    # ...same but slightly different formatting/code:
    #
    length = len(nums)
    if not length:
        return 0
    elif length == 1:
        return nums[0]
    elif length == 2:
        return max(nums[0], nums[1])

    dp = [0 for i in range(length)]

    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])
    for i in range(2, length):
        dp[i] = max( nums[i]+dp[i-2], dp[i-1])

    return dp[length-1]
```

(44) Reverse Linked List [Linked List]

Given the `head` of a singly linked list, reverse the list, and return *the reversed list.*

```
def reverseList(self, head: ListNode) -> ListNode:
    node = head
    prev = None
    while node:
        curr_next = node.next
        node.next = prev

        prev = node
        node = curr_next
    return prev
```

(50) Contains Duplicate [Array]

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

```
# solutions using 'Set'
return len(nums) > len(set(nums))

# solution iterating through sorted list
  nums.sort()
  for i in range(len(nums)-1):
      if nums[i]==nums[i+1]:
          return True
  return False
```

(51) Invert Binary Tree [Tree]

```
def invertTree(self, root: TreeNode) -> TreeNode:
    if not root:
        return None
    if root.left == None and root.right == None:
        return root
    else:
        old_left   = root.left
        root.left  = root.right
        root.right = old_left
        self.invertTree(root.left)
        self.invertTree(root.right)

    return root
```

(53) Lowest Common Ancestor [Tree]

Given a binary search tree , find the lowest common ancestor (LCA) of two given nodes.

- take advantage of tree being sorted and greater values on right and lower values on left

```
# root: 'TreeNode', p: 'TreeNode', q: 'TreeNode'
    curr = root
    while True:
        if p.val > curr.val and q.val > curr.val:
            curr = curr.right  # continue look right
        elif p.val < curr.val and q.val < curr.val:
            curr = curr.left  # continue look left
        else:
            # at 'split' meaning ones larger one smaller
            # so this is the lca
            return curr
```

(55) Valid Anagram [String]

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

```
    # s = "anagram", t = "nagaram"
    if len(s) != len(t):
        return False
    s_counter = Counter(s)
    t_counter = Counter(t)
    diff = s_counter - t_counter
    return not diff
```

(56) Missing Number [Binary]

Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

```
# solution which requires sorting
length = len(nums)
nums.sort()
for i in range(0, length + 1):
    try:
        if i != nums[i]:
            return i
    except IndexError:
        return i

# solution using sum 'trick'
nums_length = len(nums)
sum_of_range = int(nums_length * (nums_length+1) / 2)
sum_of_nums = sum(nums)
return sum_of_range - sum_of_nums
```

(69) Sum of Two Integers [Binary]

```
for i in range(32):
    # (1) if bin representations are completely opposite, XOR operation
    # will directly produce sum of numbers ( in this case carry is 0 )
    xor_sum = a ^ b

    # (2) if numbers bin representation is not completely opposite,
    # XOR will only have part of the sum and remaining will be carry;
    # carry-over can be produced by AND operation followed by left
    # shift operation. e.g. 10, 11 => 1010, 1011 => (a&b)<<1 => 10100
    carry = (a & b) << 1

    # (3) now find sum of (1) and (2)
    # i.e a is replace 'a' with XOR result
    # and 'b' is replaced wth carry result
    a = xor_sum
    b = carry
# enforce 32bit length in case of overflow
if b > 0:
```

```
    return a & 0b11111111111111111111111111111111
else:
    return a
```

(71) Longest Repeating Character Replacement [Sliding window, String]

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times. Return the length of the longest substring containing the same letter you can get after performing the above operations.

- sliding window

```
max_substring_length = 0
letter_count = {}

left = 0
for right in range(len(s)):
    letter = s[right]
    letter_count[letter] = letter_count.get(letter, 0) + 1
    substr_length = right - left + 1
    most_frequent_letter = max(letter_count.values())
    if substr_length - most_frequent_letter > k:
        current_left_letter = s[left]
        letter_count[current_left_letter] -= 1
        left += 1
        substr_length = right - left + 1

    max_substring_length = max(max_substring_length, substr_length)

return max_substring_length
```

(72) Non-overlapping Intervals [Interval]

Given an array of intervals intervals where intervals[i] = [start_i, end_i], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

- "mininum intervals you need to remove" *same as* "maximum set of non-overlapping"
- similar as maximise 'list of activities' problem (with start/end time):
  - sort the `activities` into ascending order by finishing time
  - choose any `activity` with the earliest finishing time.
  - remove from all `activities` that overlap S.

```
# e.g. intervals = [[1, 2], [2, 3], [3, 4], [1, 3]]
intervals.sort(key=lambda i: i[1])
skipped_list = []
curr_end = intervals[0][1]

for i in range(1, len(intervals)):
    interval_start = intervals[i][0]
    if (interval_start < curr_end): # check if overlap
        skipped_list.append(i)
    elif interval_start >= curr_end:
        curr_end = intervals[i][1]

return len(skipped_list)
```

(73) Subtree of another Tree [Tree]

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

```
def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
    if not p and not q: # both empty?
        return True
    if (not p or not q) or (p.val != q.val):
        return False
    return ( self.isSameTree(p.left, q.left) and
             self.isSameTree(p.right, q.right) )

def isSubtree(self, root: TreeNode, subRoot: TreeNode) -> bool:
    if not root and not subRoot: #both empty
        return True
    if not root or not subRoot:
        return False
    if self.isSameTree(root, subRoot):
        return True
    else:
        if self.isSubtree(root.left, subRoot):
            return True
        if self.isSubtree(root.right, subRoot):
            return True
    return False
```

Misc: Calculate Parity

```
def parity(number:int) -> bool:
    result = 0
    while number:
        result = result ^ (number & 1)
        number >>= 1
    return False if result else True
    if not result:
        return True
    else:
        return False
```

Misc: Delete Duplicate from sorted array - O(n) time and O(1) space

- list is sorted i.e. repeated elements must appear one-after-anothe1,
- `write_index`: advance pointer only if numbers [curr vs curr-1] are different
- if same leave for new index of next different number

```
write_index = 1
for i in range(1, len(arr)-1):
    if arr[write_index-1] != arr[i]:
        arr[write_index] = arr[i]
        write_index += 1
print(arr[0:write_index])
```