

## ASSIGNMENT-5

1. 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]`

PROGRAM:

```
def twoSum(nums, target):
    num_to_index = {} # Hash table to store indices of elements

    for i, num in enumerate(nums):
        complement = target - num
        if complement in num_to_index:
            return [num_to_index[complement], i]
        num_to_index[num] = i

nums1, target1 = [2, 7, 11, 15], 9
nums2, target2 = [3, 2, 4], 6
nums3, target3 = [3, 3], 6

print(twoSum(nums1, target1))
print(twoSum(nums2, target2))
print(twoSum(nums3, target3))
```

```
[0, 1]
[1, 2]
[0, 1]
```

OUTPUT:

TIME COMPLEXITY:  $O(n)$

2. You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1: Input: `l1 = [2,4,3]`, `l2 = [5,6,4]` Output: `[7,0,8]` Explanation:  $342 + 465 = 807$ .

Example 2: Input: `l1 = [0]`, `l2 = [0]` Output: `[0]`

Example 3: Input: `l1 = [9,9,9,9,9,9,9]`, `l2 = [9,9,9,9]` Output: `[8,9,9,9,0,0,0,1]`

PROGRAM:

class ListNode:

```

def __init__(self, val=0, next=None):
    self.val = val
    self.next = next

def addTwoNumbers(l1, l2):
    dummy_head = ListNode()
    current = dummy_head
    carry = 0

    while l1 or l2 or carry:
        sum_val = carry
        if l1:
            sum_val += l1.val
            l1 = l1.next
        if l2:
            sum_val += l2.val
            l2 = l2.next

        carry = sum_val // 10
        sum_val %= 10

        current.next = ListNode(sum_val)
        current = current.next

    return dummy_head.next

def list_to_linked_list(lst):
    dummy_head = ListNode()
    current = dummy_head
    for val in lst:
        current.next = ListNode(val)
        current = current.next
    return dummy_head.next

def linked_list_to_list(head):
    result = []
    current = head
    while current:
        result.append(current.val)
        current = current.next
    return result

l1 = list_to_linked_list([2, 4, 3])
l2 = list_to_linked_list([5, 6, 4])
print(linked_list_to_list(addTwoNumbers(l1, l2)))

l1 = list_to_linked_list([0])
l2 = list_to_linked_list([0])

```

```
print(linked_list_to_list(addTwoNumbers(l1, l2)))
```

```
[7, 0, 8]
[0]
```

OUTPUT:

TIME COMPLEXITY:  $O(\max(m,n))$

3. Longest Substring without Repeating Characters Given a string  $s$ , find the length of the longest substring without repeating characters.
- Example 1: Input:  $s = \text{"abcabcbb"}$  Output: 3 Explanation: The answer is "abc", with the length of 3.
- Example 2: Input:  $s = \text{"bbbbb"}$  Output: 1 Explanation: The answer is "b", with the length of 1.
- Example 3: Input:  $s = \text{"pwwkew"}$  Output: 3 Explanation: The answer is "wke", with the length of 3. Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

PROGRAM:

```
def lengthOfLongestSubstring(s):
    n = len(s)
    max_length = 0
    char_set = set()
    left, right = 0, 0

    while right < n:
        if s[right] not in char_set:
            char_set.add(s[right])
            max_length = max(max_length, right - left + 1)
            right += 1
        else:
            char_set.remove(s[left])
            left += 1

    return max_length

s1 = "abcabcbb"
s2 = "bbbbb"

print(lengthOfLongestSubstring(s1))
print(lengthOfLongestSubstring(s2))
```

```
3
1
```

OUTPUT:

TIME COMPLEXITY:  $O(n)$

4. Median of Two Sorted Arrays Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays. The overall run time complexity should be  $O(\log(m+n))$ .

Example 1: Input: nums1 = [1,3], nums2 = [2] Output: 2.00000 Explanation: merged array = [1,2,3] and median is 2.

Example 2: Input: nums1 = [1,2], nums2 = [3,4] Output: 2.50000 Explanation: merged array = [1,2,3,4] and median is  $(2 + 3) / 2 = 2.5$ .

PROGRAM:

```
def findMedianSortedArrays(nums1, nums2):
    m, n = len(nums1), len(nums2)

    if m > n:
        nums1, nums2, m, n = nums2, nums1, n, m

    left, right, half_len = 0, m, (m + n + 1) // 2

    while left <= right:
        i = (left + right) // 2
        j = half_len - i

        if i < m and nums2[j - 1] > nums1[i]:
            left = i + 1
        elif i > 0 and nums1[i - 1] > nums2[j]:
            right = i - 1
        else:
            if i == 0:
                max_of_left = nums2[j - 1]
            elif j == 0:
                max_of_left = nums1[i - 1]
            else:
                max_of_left = max(nums1[i - 1], nums2[j - 1])


            if (m + n) % 2 == 1:
                return max_of_left

            if i == m:
                min_of_right = nums2[j]
            elif j == n:
                min_of_right = nums1[i]
            else:
                min_of_right = min(nums1[i], nums2[j])

            return (max_of_left + min_of_right) / 2.0

nums1_1, nums2_1 = [1, 3], [2]
nums1_2, nums2_2 = [1, 2], [3, 4]
```

```
print(findMedianSortedArrays(nums1_1, nums2_1))
print(findMedianSortedArrays(nums1_2, nums2_2))
```

OUTPUT: 

TIME COMPLEXITY:  $O(\log(\min(m,n)))$

5. Longest Palindromic Substring Given a string *s*, return the longest palindromic substring in *s*.  
Example 1: Input: *s* = "babad" Output: "bab" Explanation: "aba" is also a valid answer.  
Example 2: Input: *s* = "cbbd" Output: "bb"

PROGRAM:

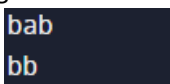
```
def longestPalindrome(s):
    n = len(s)
    start = 0
    max_length = 0
    for i in range(n):
        left = right = i
        while left >= 0 and right < n and s[left] == s[right]:
            if right - left + 1 > max_length:
                start = left
                max_length = right - left + 1
            left -= 1
            right += 1

        left = i
        right = i + 1
        while left >= 0 and right < n and s[left] == s[right]:
            if right - left + 1 > max_length:
                start = left
                max_length = right - left + 1
            left -= 1
            right += 1

    return s[start:start + max_length]
```

```
s1 = "babad"
s2 = "cbbd"
```

```
print(longestPalindrome(s1))
print(longestPalindrome(s2))
```

OUTPUT: 

TIME COMPLEXITY:  $O(n^2)$

6. Zigzag Conversion The string "PAYPALISHIRING" is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility) P A H N A P L S I I G Y I R And then read line by line: "PAHNAPLSIIGYIR" Write the code that will take a string and make this conversion given a number of rows: string convert(string s, int numRows);
- Example 1: Input: s = "PAYPALISHIRING", numRows = 3 Output: "PAHNAPLSIIGYIR"
- Example 2: Input: s = "PAYPALISHIRING", numRows = 4 Output: "PINALSIGYAHRPI"
- Explanation: P I N A L S I G Y A H R P I

PROGRAM:

```
def convert(s, numRows):
    if numRows == 1 or numRows >= len(s):
        return s

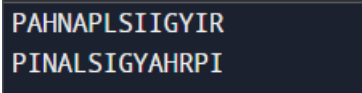
    rows = [''] * numRows
    direction = -1 # Start with upward direction
    row = 0

    for char in s:
        rows[row] += char
        if row == 0 or row == numRows - 1:
            direction *= -1 # Change direction at the top or bottom row
        row += direction

    return ''.join(rows)

s1, numRows1 = "PAYPALISHIRING", 3
s2, numRows2 = "PAYPALISHIRING", 4

print(convert(s1, numRows1))
print(convert(s2, numRows2))
```



```
PAHNAPLSIIGYIR
PINALSIGYAHRPI
```

OUTPUT:

TIME COMPLEXITY: O(n)

7. Reverse Integer Given a signed 32-bit integer x, return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0. Assume the environment does not allow you to store 64-bit integers (signed or unsigned).
- Example 1: Input: x = 123 Output: 321
- Example 2: Input: x = -123 Output: -321
- Example 3: Input: x = 120 Output: 21

PROGRAM:

```
def reverse(x):
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31
```

```

sign = 1 if x >= 0 else -1
x *= sign

reversed_x = 0
while x != 0:
    digit = x % 10
    reversed_x = reversed_x * 10 + digit
    x //= 10

reversed_x *= sign

if reversed_x < INT_MIN or reversed_x > INT_MAX:
    return 0

return reversed_x

```

```

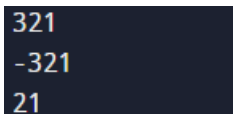
x1 = 123
x2 = -123
x3 = 120

```

```

print(reverse(x1))
print(reverse(x2))
print(reverse(x3))

```



```

321
-321
21

```

OUTPUT:

TIME COMPLEXITY:  $O(\log(x))$

8. String to Integer (atoi) Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer (similar to C/C++'s atoi function). The algorithm for myAtoi(string s) is as follows: 1. Read in and ignore any leading whitespace. 2. Check if the next character (if not already at the end of the string) is '-' or '+'. Read this character in if it is either.

Example 1: Input: s = "42" Output: 42

Example 2: Input: s = " -42" Output: -42

PROGRAM:

```

def myAtoi(s):
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    i = 0
    n = len(s)

    while i < n and s[i] == ' ':

```

```

        i += 1


    sign = 1
    if i < n and (s[i] == '-' or s[i] == '+'):
        if s[i] == '-':
            sign = -1
        i += 1

    result = 0
    while i < n and s[i].isdigit():
        digit = int(s[i])
        result = result * 10 + digit
        i += 1
    result *= sign
    if result < INT_MIN:
        return INT_MIN
    elif result > INT_MAX:
        return INT_MAX
    else:
        return result

s1 = "42"
s2 = " -42"
s3 = "4193 with words"

print(myAtoi(s1))
print(myAtoi(s2))
print(myAtoi(s3))

```



```

42
-42
4193

```

OUTPUT:

TIME COMPLEXITY:  $O(n)$

9. **Palindrome Number** Given an integer  $x$ , return true if  $x$  is a palindrome, and false otherwise.  
 Example 1: Input:  $x = 121$  Output: true Explanation: 121 reads as 121 from left to right and from right to left.  
 Example 2: Input:  $x = -121$  Output: false Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

PROGRAM:

```

def isPalindrome(x):
    # Special cases: negative integers and numbers ending with 0 (excluding 0 itself) are not
    palindromes
    if x < 0 or (x % 10 == 0 and x != 0):

```



```

    return False

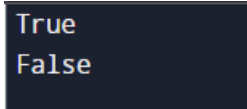
    reversed_x = 0
    original_x = x

    while x > 0:
        digit = x % 10
        reversed_x = reversed_x * 10 + digit
        x //= 10

    return original_x == reversed_x
x1 = 121
x2 = -121

print(isPalindrome(x1))
print(isPalindrome(x2))

```

OUTPUT: 

TIME COMPLEXITY:  $O(\log(x))$

10. Regular Expression Matching Given an input string  $s$  and a pattern  $p$ , implement regular expression matching with support for '.' and '\*' where: ● '.' Matches any single character. ● '\*' Matches zero or more of the preceding element. The matching should cover the entire input string (not partial).

Example 1: Input:  $s = "aa"$ ,  $p = "a"$  Output: false Explanation: "a" does not match the entire string "aa".

Example 2: Input:  $s = "aa"$ ,  $p = "a*"$  Output: true Explanation: '\*' means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

PROGRAM:

```

def isMatch(s, p):
    m, n = len(s), len(p)
    dp = [[False] * (n + 1) for _ in range(m + 1)]
    dp[0][0] = True

    for j in range(1, n + 1):
        if p[j - 1] == '*':
            dp[0][j] = dp[0][j - 2]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if p[j - 1] == '.' or p[j - 1] == s[i - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            elif p[j - 1] == '*':

```

```
dp[i][j] = dp[i][j - 2] or (dp[i - 1][j] and (s[i - 1] == p[j - 2] or p[j - 2] == '.'))
```

```
return dp[m][n]
```

```
s1, p1 = "aa", "a"
```

```
s2, p2 = "aa", "a*"
```

```
print(isMatch(s1, p1))
```

```
print(isMatch(s2, p2))
```

```
False
True
```

OUTPUT:

TIME COMPLEXITY:  $O(m*n)$

11. Container With Most Water You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]). Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. Notice that you may not slant the container.

Example 1: Input: height = [1,8,6,2,5,4,8,3,7] Output: 49 Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49. Example 2: Input: height = [1,1] Output: 1

PROGRAM:

```
def maxArea(height):
```

```
    max_area = 0
```

```
    left, right = 0, len(height) - 1
```

```
    while left < right:
```

```
        area = min(height[left], height[right]) * (right - left)
```

```
        max_area = max(max_area, area)
```

```
        if height[left] < height[right]:
```

```
            left += 1
```

```
        else:
```

```
            right -= 1
```

```
    return max_area
```

```
height1 = [1, 8, 6, 2, 5, 4, 8, 3, 7]
```

```
height2 = [1, 1]
```

```
print(maxArea(height1))
```

```
print(maxArea(height2))
```



```
49
1
```

OUTPUT:

TIME COMPLEXITY:  $O(1)$

12. Integer to Roman Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M. Symbol Value I 1 V 5 X 10 L 50 C 100 D 500 M 1000 For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II. Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

Example 1: Input: num = 3 Output: "III" Explanation: 3 is represented as 3 ones.

Example 2: Input: num = 58 Output: "LVIII" Explanation: L = 50, V = 5, III = 3.

PROGRAM:

```
def intToRoman(num):
    roman_map = {
        1: 'I', 4: 'IV', 5: 'V', 9: 'IX',
        10: 'X', 40: 'XL', 50: 'L', 90: 'XC',
        100: 'C', 400: 'CD', 500: 'D', 900: 'CM',
        1000: 'M'
    }

    roman_numeral = ""

    for value, symbol in sorted(roman_map.items(), reverse=True):
        while num >= value:
            roman_numeral += symbol
            num -= value

    return roman_numeral

num1 = 3
num2 = 58

print(intToRoman(num1))
print(intToRoman(num2))
```



```
III
LVIII
```

OUTPUT:

TIME COMPLEXITY:  $O(1)$

13. Roman to Integer Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M. Symbol Value I 1 V 5 X 10 L 50 C 100 D 500 M 1000 For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II. Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used: ● I can be placed before V (5) and X (10) to make 4 and 9. ● X can be placed before L (50) and C (100) to make 40 and 90. ● C can be placed before D (500) and M (1000) to make 400 and 900. Given a roman numeral, convert it to an integer.

Example 1: Input: s = "III" Output: 3 Explanation: III = 3.

Example 2: Input: s = "LVIII" Output: 58 Explanation: L = 50, V= 5, III = 3

PROGRAM:

```
def romanToInt(s):
    roman_map = {
        'I': 1, 'V': 5, 'X': 10, 'L': 50,
        'C': 100, 'D': 500, 'M': 1000
    }


    prev_value = 0
    total = 0

    for char in s:
        value = roman_map[char]
        if value > prev_value:
            total += value - 2 * prev_value
        else:
            total += value
        prev_value = value

    return total

s1 = "III"
s2 = "LVIII"

print(romanToInt(s1))
print(romanToInt(s2))
```

OUTPUT: 

TIME COMPLEXITY: O(n)

14. Longest Common Prefix Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string "".

Example 1: Input: strs = ["flower", "flow", "flight"] Output: "fl"

Example 2: Input: strs = ["dog", "racecar", "car"] Output: "" Explanation: There is no common prefix among the input strings.

PROGRAM:

```
def longestCommonPrefix(strs):
    if not strs:
        return ""

    for i, char in enumerate(strs[0]):
        # Check if the character exists in all other strings
        for string in strs[1:]:
            if i >= len(string) or string[i] != char:
                return strs[0][:i]

    return strs[0]
```

```
strs1 = ["flower", "flow", "flight"]
strs2 = ["dog", "racecar", "car"]
```

```
print(longestCommonPrefix(strs1))
print(longestCommonPrefix(strs2))
```



OUTPUT:

TIME COMPLEXITY:  $O(n*m)$

### 15. 3Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that  $i \neq j$ ,  $i \neq k$ , and  $j \neq k$ , and  $nums[i] + nums[j] + nums[k] == 0$ . Notice that the solution set must not contain duplicate triplets.

Example 1: Input: nums = [-1,0,1,2,-1,-4] Output: [[-1,-1,2],[-1,0,1]] Explanation:  $nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0$ .  $nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0$ .  $nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0$ . The distinct triplets are [-1,0,1] and [-1,-1,2]. Notice that the order of the output and the order of the triplets does not matter.

Example 2: Input: nums = [0,1,1] Output: [] Explanation: The only possible triplet does not sum up to 0

PROGRAM:

```
def threeSum(nums):
    nums.sort() # Sort the array
    n = len(nums)
    triplets = []

    for i in range(n - 2):

        if i > 0 and nums[i] == nums[i - 1]:
            continue
```

```

left, right = i + 1, n - 1

while left < right:
    total = nums[i] + nums[left] + nums[right]
    if total == 0:
        triplets.append([nums[i], nums[left], nums[right]])
        # Avoid duplicates
        while left < right and nums[left] == nums[left + 1]:
            left += 1
        while left < right and nums[right] == nums[right - 1]:
            right -= 1
        left += 1
        right -= 1
    elif total < 0:
        left += 1
    else:
        right -= 1

return triplets

nums1 = [-1, 0, 1, 2, -1, -4]
nums2 = [0, 1, 1]

print(threeSum(nums1))
print(threeSum(nums2))

```

```

[[-1, -1, 2], [-1, 0, 1]]
[]

```

OUTPUT:

TIME COMPLEXITY:  $O(n^2)$

16. 3Sum Closest Given an integer array `nums` of length `n` 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 1: Input: `nums = [-1,2,1,-4]`, `target = 1` Output: 2 Explanation: The sum that is closest to the target is 2.  $(-1 + 2 + 1 = 2)$ .
- Example 2: Input: `nums = [0,0,0]`, `target = 1` Output: 0 Explanation: The sum that is closest to the target is 0.  $(0 + 0 + 0 = 0)$ .

PROGRAM:

```

def threeSumClosest(nums, target):
    nums.sort()
    n = len(nums)
    closest_sum = float('inf')

```

```

for i in range(n - 2):
    left, right = i + 1, n - 1

    while left < right:
        total = nums[i] + nums[left] + nums[right]
        if abs(total - target) < abs(closest_sum - target):
            closest_sum = total


        if total < target:
            left += 1
        elif total > target:
            right -= 1
        else:
            return target

    return closest_sum

nums1, target1 = [-1, 2, 1, -4], 1
nums2, target2 = [0, 0, 0], 1

print(threeSumClosest(nums1, target1))
print(threeSumClosest(nums2, target2))

```

OUTPUT: 

TIME COMPLEXITY:  $O(n^2)$

#### 17. Letter Combinations of a Phone Number

Given a string containing digits from 2-9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order. A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1: Input: digits = "23" Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]

Example 2: Input: digits = "" Output: []

PROGRAM:

```

def letterCombinations(digits):
    if not digits:
        return []
    mappings = {
        '2': 'abc',
        '3': 'def',
        '4': 'ghi',
        '5': 'jkl',
        '6': 'mno',
        '7': 'pqrs',
        '8': 'tuv',

```

```

    '9': 'wxyz'
}

def backtrack(combination, next_digits):
    if not next_digits:
        output.append(combination)
    else:
        for letter in mappings[next_digits[0]]:
            backtrack(combination + letter, next_digits[1:])

output = []
backtrack("", digits)
return output

print(letterCombinations("23"))
print(letterCombinations(""))

```

```

['ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd',
 'ce', 'cf']
[]

```

OUTPUT:

TIME COMPLEXITY:  $O(4^n)$

18. 4Sum Given an array nums of n integers, return an array of all the unique quadruplets [nums[a], nums[b], nums[c], nums[d]] such that: •  $0 \leq a, b, c, d < n$  • a, b, c, and d are distinct. •  $\text{nums}[a] + \text{nums}[b] + \text{nums}[c] + \text{nums}[d] == \text{target}$  You may return the answer in any order.

Example 1: Input: nums = [1,0,-1,0,-2,2], target = 0 Output: [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

Example 2: Input: nums = [2,2,2,2,2], target = 8 Output: [[2,2,2,2]]

PROGRAM:

```

def fourSum(nums, target):
    nums.sort()
    n = len(nums)
    quadruplets = []

    for i in range(n - 3):
        if i > 0 and nums[i] == nums[i - 1]:
            continue
        for j in range(i + 1, n - 2):
            if j > i + 1 and nums[j] == nums[j - 1]:
                continue
            left, right = j + 1, n - 1
            while left < right:

```



```

total = nums[i] + nums[j] + nums[left] + nums[right]
if total == target:
    quadruplets.append([nums[i], nums[j], nums[left], nums[right]])

    while left < right and nums[left] == nums[left + 1]:
        left += 1
    while left < right and nums[right] == nums[right - 1]:
        right -= 1
    left += 1
    right -= 1
elif total < target:
    left += 1
else:
    right -= 1

return quadruplets

print(fourSum([1,0,-1,0,-2,2], 0))
print(fourSum([2,2,2,2,2], 8))

```

```

[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
[[2, 2, 2, 2]]

```

OUTPUT:

TIME COMPLEXITY:  $O(n^3 + n \log n)$

19. Remove Nth Node From End of List Given the head of a linked list, remove the nth node from the end of the list and return its head.

Example 1: Input: head = [1,2,3,4,5], n = 2 Output: [1,2,3,5]

Example 2: Input: head = [1], n = 1 Output: [] Example 3: Input: head = [1,2], n = 1 Output: [1]

PROGRAM:

```

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def removeNthFromEnd(head, n):
    dummy = ListNode(0)
    dummy.next = head
    first = dummy
    second = dummy
    for _ in range(n + 1):
        first = first.next
    while first is not None:
        first = first.next
        second = second.next
    second.next = second.next.next

```

```

return dummy.next

def printLinkedList(head):
    current = head
    while current is not None:
        print(current.val, end=" -> ")
        current = current.next
    print("None")

head1 = ListNode(1)
head1.next = ListNode(2)
head1.next.next = ListNode(3)
head1.next.next.next = ListNode(4)
head1.next.next.next.next = ListNode(5)
n1 = 2
printLinkedList(removeNthFromEnd(head1, n1))

head2 = ListNode(1)
n2 = 1
printLinkedList(removeNthFromEnd(head2, n2))

```

```

1 -> 2 -> 3 -> 5 -> None
None

```

OUTPUT:

TIME COMPLEXITY:  $O(n)$

20. Valid Parentheses Given a string *s* containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid. An input string is valid if: 1. Open brackets must be closed by the same type of brackets. 2. Open brackets must be closed in the correct order. 3. Every close bracket has a corresponding open bracket of the same type.
- Example 1: Input: *s* = "()" Output: true
- Example 2: Input: *s* = "()[]{}" Output: true

PROGRAM:

```

def isValid(s):
    stack = []
    mapping = {"(": ")", "{": "}", "[": ""]

    for char in s:
        if char in mapping:
            top_element = stack.pop() if stack else '#'
            if mapping[char] != top_element:
                return False
        else:
            stack.append(char)

```

```
return not stack
```

```
s1 = "()"
print(isValid(s1))
```

```
s2 = "()[]{}"
print(isValid(s2))
```



```
True
True
```

OUTPUT:

TIME COMPLEXITY:  $O(n)$