

CSA0674-DESIGN ANALYSIS AND ALGORITHM

ASSIGNMENT-05

KORNANA GOUTHAM

192311169

DEPT: CSE(General)

1. Two Sum 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]`.

```
1 def two_sum(nums, target):  
2     # Initialize an empty hash map  
3     num_to_index = {}  
4  
5     # Iterate through the array  
6     for index, num in enumerate(nums):  
7         # Calculate the complement  
8         complement = target - num  
9  
10        # Check if the complement is in the hash map  
11        if complement in num_to_index:  
12            # If found, return the indices  
13            return [num_to_index[complement], index]  
14  
15        # Otherwise, add the current number and its index to the hash map  
16        num_to_index[num] = index  
17  
18 # Example usage  
19 nums = [2, 7, 11, 15]  
20 target = 9  
21 print(two_sum(nums, target)) # Output: [0, 1]  
22
```

2. Add Two Numbers 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: $342 + 465 = 807$

```
1 class ListNode:
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5
6 def add_two_numbers(l1, l2):
7     # Initialize a dummy node and a current node pointer
8     dummy = ListNode()
9     current = dummy
10    carry = 0
11
12    # Traverse both linked lists
13    while l1 or l2 or carry:
14        # Get values from the current nodes of l1 and l2, if t
15        val1 = l1.val if l1 else 0
16        val2 = l2.val if l2 else 0
17
18        # Calculate the sum and carry
19        total = val1 + val2 + carry
20        carry = total // 10
21        new_val = total % 10
22
23        # Create a new node with the sum's single digit
24        current.next = ListNode(new_val)
25        current = current.next
```

```

7         # Move to the next nodes in l1 and l2, if they exist
8         if l1: l1 = l1.next
9         if l2: l2 = l2.next
10
11     # The result is in the next node of dummy (since dummy is a placeholder)
12     return dummy.next
13
14 # Example usage
15 def create_linked_list(elements):
16     head = ListNode(elements[0])
17     current = head
18     for element in elements[1:]:
19         current.next = ListNode(element)
20         current = current.next
21     return head
22
23 def print_linked_list(node):
24     while node:
25         print(node.val, end=" -> " if node.next else "\n")
26         node = node.next
27
28 l1 = create_linked_list([2, 4, 3])
29 l2 = create_linked_list([5, 6, 4])
30 result = add_two_numbers(l1, l2)
31 print_linked_list(result) # Output: 7 -> 0 -> 8

```

Output

```
7 -> 0 -> 8
```

```
=== Code Execution Successful ===
```

3. Longest Substring without Repeating Characters Given a string *s*, find the length of the longest substring without repeating characters.

Example 1: Input: *s* = "abcabcbb"

Output: 3

Explanation: The answer is "abc", with the length of 3.

```
1- def length_of_longest_substring(s: str) -> int:
2-     # Dictionary to store the most recent index of each character
3-     char_index_map = {}
4-     max_length = 0
5-     left = 0
6-
7-     # Iterate over the string with the right pointer
8-     for right, char in enumerate(s):
9-         # If the character is already in the dictionary and its index is within
           the current window
10-         if char in char_index_map and char_index_map[char] >= left:
11-             # Move the left pointer to the right of the last index of the
               character
12-             left = char_index_map[char] + 1
13-
14-         # Update the dictionary with the current character's index
15-         char_index_map[char] = right
16-
17-         # Calculate the current window length and update the maximum length
18-         max_length = max(max_length, right - left + 1)
19-
20-     return max_length
21-
22- # Example usage
23- s = "abcabcbb"
24- print(length_of_longest_substring(s)) # Output: 3
```

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.

```

def findMedianSortedArrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1
    m, n = len(nums1), len(nums2)
    imin, imax, half_len = 0, m, (m + n + 1) // 2
    while imin <= imax:
        i = (imin + imax) // 2
        j = half_len - i
        if i < m and nums1[i] < nums2[j - 1]:
            imin = i + 1
        elif i > 0 and nums1[i - 1] > nums2[j]:
            imax = 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, 3]
nums2 = [2]

```

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.

```

1 def longestPalindrome(s: str) -> str:
2     n = len(s)
3     if n < 2:
4         return s
5     dp = [[False] * n for _ in range(n)]
6     start = 0
7     max_length = 1
8     for i in range(n):
9         dp[i][i] = True
10    for i in range(n - 1):
11        if s[i] == s[i + 1]:
12            dp[i][i + 1] = True
13            start = i
14            max_length = 2
15    for length in range(3, n + 1):
16        for i in range(n - length + 1):
17            j = i + length - 1
18            if s[i] == s[j] and dp[i + 1][j - 1]:
19                dp[i][j] = True
20                start = i
21                max_length = length
22
23    return s[start:start + max_length]
24
25 # Example usage
26 s = "babad"

```

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);

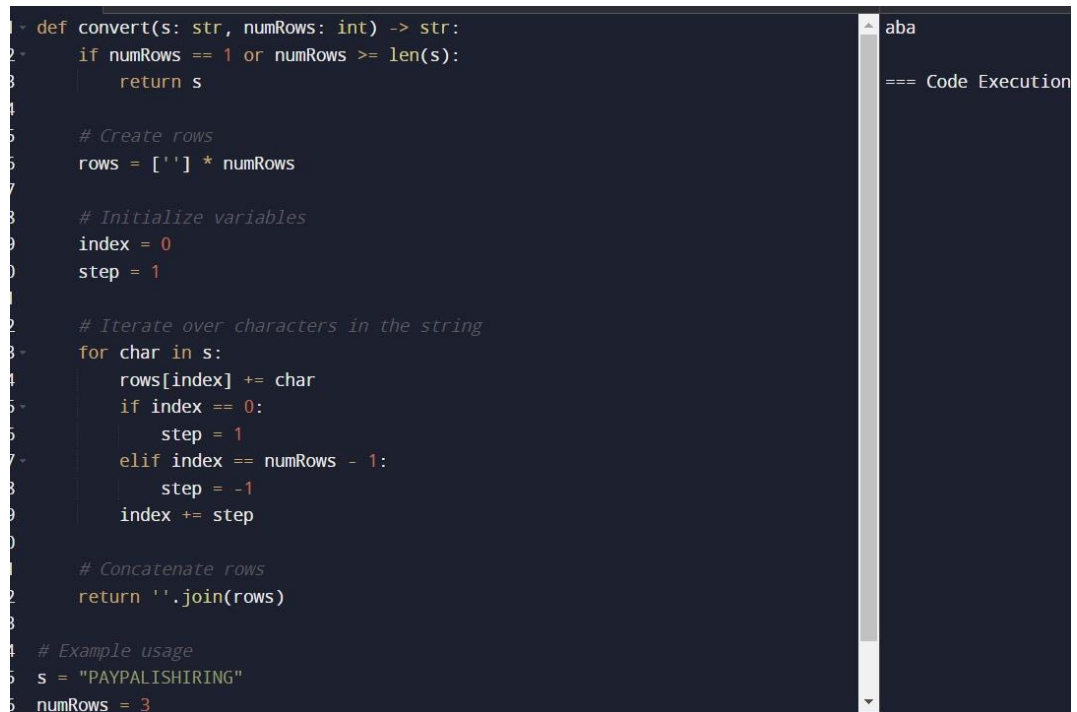
Example 1: Input: s = "PAYPALISHIRING", numRows = 3

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



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

    # Create rows
    rows = [''] * numRows

    # Initialize variables
    index = 0
    step = 1

    # Iterate over characters in the string
    for char in s:
        rows[index] += char
        if index == 0:
            step = 1
        elif index == numRows - 1:
            step = -1
        index += step

    # Concatenate rows
    return ''.join(rows)

# Example usage
s = "PAYPALISHIRING"
numRows = 3
```

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

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

    result = 0
    negative = x < 0
    x = abs(x)

    while x != 0:
        digit = x % 10
        x //= 10

        # Check for overflow
        if result > (INT_MAX - digit) // 10:
            return 0

        result = result * 10 + digit

    return -result if negative else result

# Example usage
x = 123
print(reverse(x)) # Output: 321

x = -123
print(reverse(x)) # Output: -321
```

321
-321
0
=== Code Execution Su

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. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present.
3. Read in next the characters until the next non-digit character or the end of the input is reached. The rest of the string is ignored.
4. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then the integer is 0. Change the sign as necessary (from step 2).
5. If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp the integer so that it remains in the range. Specifically, integers less than -231 should be clamped to -231, and integers greater than 231 - 1 should be clamped to 231 - 1.

6. Return the integer as the final result.

Note:

- Only the space character ' ' is considered a whitespace character.
- Do not ignore any characters other than the leading whitespace or the rest of the string after the digits.

Example 1:

Input: s = "42"

Output: 42

Explanation: The underlined characters are what is read in, the caret is the current reader position.

Step 1: "42" (no characters read because there is no leading whitespace)

^

Step 2: "42" (no characters read because there is neither a '-' nor '+')

^

Step 3: "42" ("42" is read in)

^

The parsed integer is 42.

Since 42 is in the range $[-2^{31}, 2^{31} - 1]$, the final result is 42.

Example 2:

Input: s = " -42"

Output: -42

Explanation:

Step 1: " -42" (leading whitespace is read and ignored)

^

Step 2: " -42" ('-' is read, so the result should be negative)

^

Step 3: " -42" ("42" is read in)

^

The parsed integer is -42.

Since -42 is in the range $[-231, 231 - 1]$, the final result is -42.

Example 3:

Input: `s = "4193 with words"`

Output: 4193

Explanation:

Step 1: "4193 with words" (no characters read because there is no leading whitespace)

^

Step 2: "4193 with words" (no characters read because there is neither a '-' nor '+')

^

Step 3: "4193 with words" ("4193" is read in; reading stops because the next character is a non-digit)

^

The parsed integer is 4193.

Since 4193 is in the range $[-231, 231 - 1]$, the final result is 4193.

Constraints:

- $0 \leq s.length \leq 200$
- `s` consists of English letters (lower-case and upper-case), digits (0-9), '-', '+', and '.'.

```
main.py  [ ] [ ] Save Run Output
1- def myAtoi(s: str) -> int:
2-     INT_MAX = 2**31 - 1
3-     INT_MIN = -2**31
4-     i = 0
5-     n = len(s)
6-     while i < n and s[i] == ' ':
7-         i += 1
8-     sign = 1
9-     if i < n and (s[i] == '-' or s[i] == '+'):
10-         if s[i] == '-':
11-             sign = -1
12-         i += 1
13-     num = 0
14-     while i < n and s[i].isdigit():
15-         num = num * 10 + int(s[i])
16-         i += 1
17-     num *= sign
18-     if num < INT_MIN:
19-         return INT_MIN
20-     if num > INT_MAX:
21-         return INT_MAX
22-     return num
23- # test examples
24- print(myAtoi("42"))           # Output: 42
25- print(myAtoi("   -42"))       # Output: -42
26- print(myAtoi("4193 with words")) # Output: 4193
```

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.

Example 3:

Input: x = 10

Output: false

Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

Constraints:

- $-231 \leq x \leq 231 - 1$

main.py	Output
<pre>1 def isPalindrome(x: int) -> bool: 2 if x < 0: 3 return False 4 return str(x) == str(x)[::-1] 5 6 # Test examples 7 print(isPalindrome(121)) # Output: True 8 print(isPalindrome(-121)) # Output: False 9 print(isPalindrome(10)) # Output: False 10</pre>	<pre>True False False === Code Execution Successful ===</pre>

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".

Example 3:

Input: $s = "ab"$, $p = ".*"$

Output: true

Explanation: ".*" means "zero or more (*) of any character (.)".

Constraints:

- $1 \leq s.length \leq 20$

- $1 \leq p.length \leq 30$
- s contains only lowercase English letters.
- p contains only lowercase English letters, '.', and '*'.
- It is guaranteed for each appearance of the character '*', there will be a previous valid character to match.

main.py	Output
<pre>def isMatch(s: str, p: str) -> bool: import re return bool(re.fullmatch(p, s)) # Test examples print(isMatch("aa", "a")) # Output: False print(isMatch("aa", "a*")) # Output: True print(isMatch("ab", ".a")) # Output: True</pre>	<pre>False True True === Code Execution Successful ===</pre>

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

Constraints:

- $n == height.length$

- $2 \leq n \leq 105$
- $0 \leq \text{height}[i] \leq 104$

```

main.py
1 def maxArea(height: list[int]) -> int:
2     left, right = 0, len(height) - 1
3     max_area = 0
4     while left < right:
5         max_area = max(max_area, min(height[left], height[right]) * (right - left))
6         if height[left] < height[right]:
7             left += 1
8         else:
9             right -= 1
10    return max_area
1
2 # Test examples
3 print(maxArea([1,8,6,2,5,4,8,3,7])) # Output: 49
4 print(maxArea([1,1])) # Output: 1
5

```

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:

- 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 an integer, convert it to a roman numeral.

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.

Example 3: I

9

V

13

X

13

Input: num = 1994

Output: "MCMXCIV"

Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints:

- $1 \leq \text{num} \leq 3999$

```

main.py
1 def intToRoman(num: int) -> str:
2     val = [
3         1000, 900, 500, 400,
4         100, 90, 50, 40,
5         10, 9, 5, 4,
6         1
7     ]
8     syb = [
9         "M", "CM", "D", "CD",
10        "C", "XC", "L", "XL",
11        "X", "IX", "V", "IV",
12        "I"
13    ]
14    roman = ''
15    for i in range(len(val)):
16        while num >= val[i]:
17            roman += syb[i]
18            num -= val[i]
19    return roman
20
21 # Test examples
22 print(intToRoman(3)) # Output: "III"
23 print(intToRoman(58)) # Output: "LVIII"
24 print(intToRoman(1994)) # Output: "MCMXCIV"
25

```

Output

```

III
LVIII
MCMXCIV

=== Code Execution

```

13. Roman to Integer

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol Value

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.

Example 3:

Input: s = "MCMXCIV"

Output: 1994

Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

Constraints:

- $1 \leq s.length \leq 15$
- s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
- It is guaranteed that s is a valid roman numeral in the range [1, 3999].

main.py	Output
<pre>1- def romanToInt(s: str) -> int: 2 roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000} 3 total = 0 4 prev_value = 0 5 for char in s[::-1]: 6 value = roman[char] 7 if value < prev_value: 8 total -= value 9 else: 10 total += value 11 prev_value = value 12 return total 13 14 # Test examples 15 print(romanToInt("III")) # Output: 3 16 print(romanToInt("LVIII")) # Output: 58 17 print(romanToInt("MCMXCIV")) # Output: 1994 18</pre>	<pre>3 58 1994 === Code Execution Success</pre>

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.

Constraints:

- $1 \leq \text{strs.length} \leq 200$
- $0 \leq \text{strs}[i].\text{length} \leq 200$
- $\text{strs}[i]$ consists of only lowercase English letters.

main.py	Save	Run	Output
<pre> 1 def longestCommonPrefix(strs: list[str]) -> str: 2 if not strs: 3 return "" 4 shortest = min(strs, key=len) 5 for i, char in enumerate(shortest): 6 for other in strs: 7 if other[i] != char: 8 return shortest[:i] 9 return shortest 10 11 # Test examples 12 print(longestCommonPrefix(["flower", "flow", "flight"])) # Output: "fl" 13 print(longestCommonPrefix(["dog", "racecar", "car"])) # Output: "" 14 </pre>			<pre> fl === Code Execution S </pre>

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.

Example 3:

Input: `nums = [0,0,0]`

Output: `[[0,0,0]]`

Explanation: The only possible triplet sums up to 0.

Constraints:

- $3 \leq \text{nums.length} \leq 3000$
- $-105 \leq \text{nums}[i] \leq 105$

```
main.py  Save Run Output
3 result = []
4 for i in range(len(nums) - 2):
5     if i > 0 and nums[i] == nums[i - 1]:
6         continue
7     left, right = i + 1, len(nums) - 1
8     while left < right:
9         total = nums[i] + nums[left] + nums[right]
10        if total < 0:
11            left += 1
12        elif total > 0:
13            right -= 1
14        else:
15            result.append([nums[i], nums[left], nums[right]])
16            while left < right and nums[left] == nums[left + 1]:
17                left += 1
18            while left < right and nums[right] == nums[right - 1]:
19                right -= 1
20            left += 1
21            right -= 1
22    return result
23
24 # Test examples
25 print(threeSum([-1, 0, 1, 2, -1, -4])) # Output: [[-1, -1, 2], [-1, 0, 1]]
26 print(threeSum([0, 1, 1]))           # Output: []
27 print(threeSum([0, 0, 0]))           # Output: [[0, 0, 0]]
28
```

Output

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

=== Code Execution Successful ===
```

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:

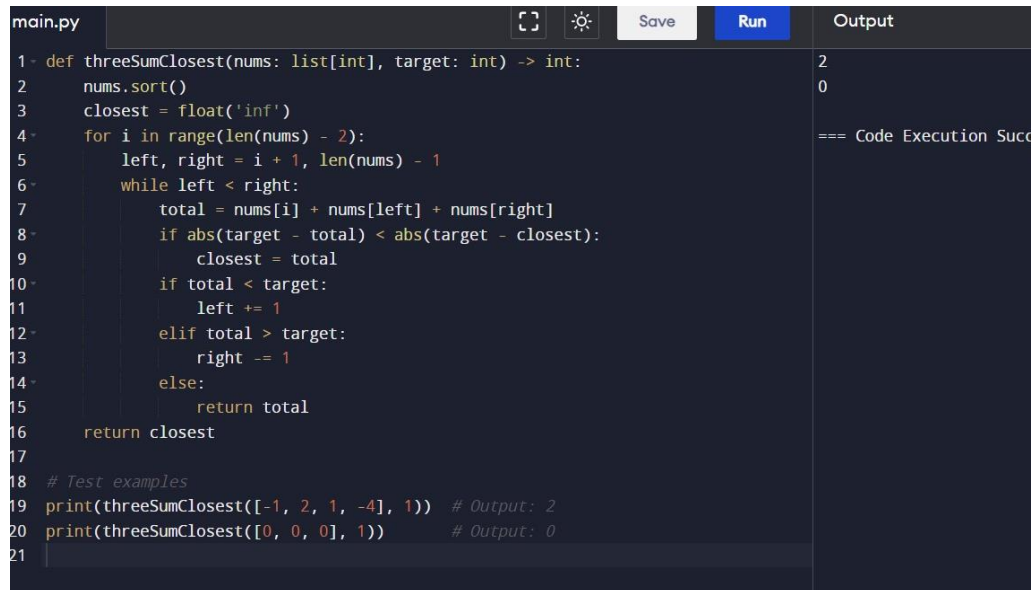
Input: `nums = [0,0,0]`, `target = 1`

Output: 0

Constraints:

Explanation: The sum that is closest to the target is 0. ($0 + 0 + 0 = 0$).

- $3 \leq \text{nums.length} \leq 500$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-104 \leq \text{target} \leq 104$



```
main.py  [Icons] Save Run Output
1 def threeSumClosest(nums: list[int], target: int) -> int:
2     nums.sort()
3     closest = float('inf')
4     for i in range(len(nums) - 2):
5         left, right = i + 1, len(nums) - 1
6         while left < right:
7             total = nums[i] + nums[left] + nums[right]
8             if abs(target - total) < abs(target - closest):
9                 closest = total
10            if total < target:
11                left += 1
12            elif total > target:
13                right -= 1
14            else:
15                return total
16        return closest
17
18 # Test examples
19 print(threeSumClosest([-1, 2, 1, -4], 1)) # Output: 2
20 print(threeSumClosest([0, 0, 0], 1))    # Output: 0
21
```

2
0
=== Code Execution Succ

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: []

Example 3:

Input: digits = "2"

Constraints:

Output: ["a","b","c"]

- $0 \leq \text{digits.length} \leq 4$
- $\text{digits}[i]$ is a digit in the range ['2', '9'].

```
main.py Save Run Output
1 def letterCombinations(digits: str) -> list[str]:
2     if not digits:
3         return []
4     phone_map = {
5         '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl', '6': 'mno',
6         '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
7     }
8     def backtrack(combination, next_digits):
9         if not next_digits:
10             output.append(combination)
11         else:
12             for letter in phone_map[next_digits[0]]:
13                 backtrack(combination + letter, next_digits[1:])
14
15     output = []
16     backtrack("", digits)
17     return output
18
19 # Test examples
20 print(letterCombinations("23")) # Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
21 print(letterCombinations("")) # Output: []
22 print(letterCombinations("2")) # Output: ["a","b","c"]
23
```

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]]`

Constraints:

- $1 \leq \text{nums.length} \leq 200$

- $-109 \leq \text{nums}[i] \leq 109$
- $-109 \leq \text{target} \leq 109$

The screenshot shows a Python IDE with a file named 'main.py'. The code defines a function 'fourSum' that takes a list of numbers and a target value. It sorts the numbers and uses a four-pointer approach to find all unique quadruplets that sum to the target. The output window shows the result of the function call: '[[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]' and a message '=== Code Execution Successful ==='.

```

1 def fourSum(nums, target):
2     nums.sort()
3     n, result = len(nums), []
4
5     for i in range(n - 3):
6         if i > 0 and nums[i] == nums[i - 1]:
7             continue
8         for j in range(i + 1, n - 2):
9             if j > i + 1 and nums[j] == nums[j - 1]:
10                continue
11            left, right = j + 1, n - 1
12            while left < right:
13                sum = nums[i] + nums[j] + nums[left] + nums[right]
14                if sum == target:
15                    result.append([nums[i], nums[j], nums[left], nums[right]])
16                    left += 1
17                    right -= 1
18                    while left < right and nums[left] == nums[left - 1]:
19                        left += 1
20                    while left < right and nums[right] == nums[right + 1]:
21                        right -= 1
22                elif sum < target:
23                    left += 1
24                else:
25                    right -= 1
26    return result

```

Output: [[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
 === Code Execution Successful ===

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]

Constraints:

- The number of nodes in the list is sz.
- $1 \leq \text{sz} \leq 30$

- $0 \leq \text{Node.val} \leq 100$

- $1 \leq n \leq \text{sz}$

```
main.py  [ ] [ ] Save Run Output
1 class ListNode:
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5
6 def removeNthFromEnd(head: ListNode, n: int) -> ListNode:
7     dummy = ListNode(0, head)
8     first = second = dummy
9     for _ in range(n + 1):
10         first = first.next
11
12     while first:
13         first = first.next
14         second = second.next
15
16     second.next = second.next.next
17     return dummy.next
18
19 # Helper function to create and print a linked list
20 def create_linked_list(arr):
21     head = ListNode(arr[0])
22     current = head
23     for val in arr[1:]:
24         current.next = ListNode(val)
25         current = current.next
26     return head
```

1 -> 2 -> 3 -> 5
=== Code Execution Successful ===


```

14     second = second.next
15
16     second.next = second.next.next
17     return dummy.next
18
19 # Helper function to create and print a linked list
20 def create_linked_list(arr):
21     head = ListNode(arr[0])
22     current = head
23     for val in arr[1:]:
24         current.next = ListNode(val)
25         current = current.next
26     return head
27
28 def print_linked_list(head):
29     current = head
30     while current:
31         print(current.val, end=" -> " if current.next else "\n")
32         current = current.next
33
34 # Example usage
35 head = create_linked_list([1, 2, 3, 4, 5])
36 n = 2
37 new_head = removeNthFromEnd(head, n)
38 print_linked_list(new_head)

```

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 Example

3:

Input: s = "()"

Output: false Constraints:

- $1 \leq s.length$

main.py	Save	Run	Output
<pre>1 def isValid(s: str) -> bool: 2 stack = [] 3 mapping = {'(': '(', ')': '}', '{': '{', '}': '[]'} 4 5 for char in s: 6 if char in mapping: 7 top_element = stack.pop() if stack else '#' 8 if mapping[char] != top_element: 9 return False 10 else: 11 stack.append(char) 12 13 return not stack 14 15 # Example usage 16 s1 = "()" 17 s2 = "()[]{}" 18 s3 = "]" 19 20 print(isValid(s1)) # Output: True 21 print(isValid(s2)) # Output: True 22 print(isValid(s3)) # Output: False 23</pre>			<pre>True True False === Code Execution</pre>