

# ASSIGNMENT-2

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:

```
main.py
1 def maxArea(height):
2     left, right = 0, len(height) - 1
3     max_area = 0
4
5     while left < right:
6         width = right - left
7         min_height = min(height[left], height[right])
8         current_area = width * min_height
9         max_area = max(max_area, current_area)
10        if height[left] < height[right]:
11            left += 1
12        else:
13            right -= 1
14
15    return max_area
16 height1 = [1, 8, 6, 2, 5, 4, 8, 3, 7]
17 height2 = [1, 1]
18 print(maxArea(height1)) # Output: 49
19 print(maxArea(height2)) # Output: 1
20
```

Output:

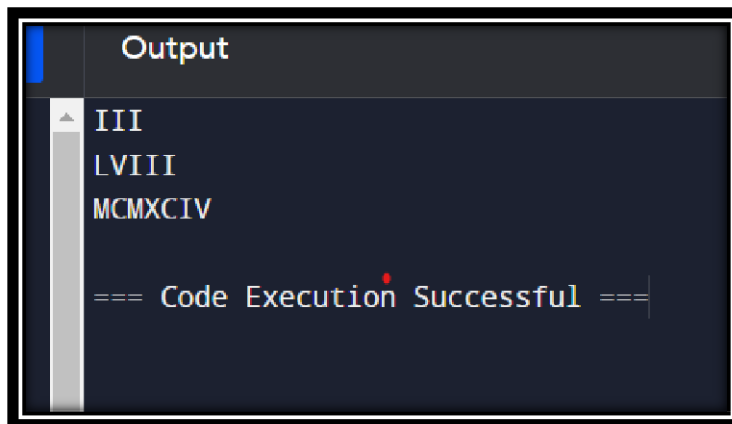
```
49
1
=== Code Execution Successful ===
```

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: Input: num = 1994 Output: "MCMXCIV" Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

### Program:

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

### Output:



```
Output
III
LVIII
MCMXCIV
=== Code Execution Successful ===
```

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. Example 3: Input: s = "MCMXCIV" Output: 1994

Program:

```
main.py
1 def romanToInt(s):
2     # Define the Roman numeral symbols and their corresponding values
3     roman_to_int = {
4         'I': 1, 'V': 5, 'X': 10, 'L': 50,
5         'C': 100, 'D': 500, 'M': 1000
6     }
7
8     # Initialize the integer value
9     total = 0
10    i = 0
11
12    while i < len(s):
13        # Check if this is a subtractive combination
14        if i + 1 < len(s) and roman_to_int[s[i]] < roman_to_int[s[i + 1]]:
15            total += roman_to_int[s[i + 1]] - roman_to_int[s[i]]
16            i += 2 # Move past this pair of characters
17        else:
18            total += roman_to_int[s[i]]
19            i += 1
20
21    return total
22
23 # Example usage
24 print(romanToInt("III")) # Output: 3
25 print(romanToInt("LVIII")) # Output: 58
26 print(romanToInt("MCMXCIV")) # Output: 1994
```

Output:

```
III
LVIII
MCMXCIV

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

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:

```
1 def longestCommonPrefix(strs):
2     if not strs:
3         return ""
4
5     # The first string is taken as the reference
6     prefix = strs[0]
7
8     # Iterate over the other strings
9     for s in strs[1:]:
10        # Compare the prefix with each string and reduce it if necessary
11        while not s.startswith(prefix):
12            prefix = prefix[:-1]
13            if not prefix:
14                return ""
15
16    return prefix
17
18 # Example usage
19 print(longestCommonPrefix(["flower", "flow", "flight"])) # Output: "fl"
20 print(longestCommonPrefix(["dog", "racecar", "car"]))    # Output: ""
21
```

### Output:

```
fl
```

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

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:  $\text{nums}[0] + \text{nums}[1] + \text{nums}[2] = (-1) + 0 + 1 = 0$ .  $\text{nums}[1] + \text{nums}[2] + \text{nums}[4] = 0 + 1 + (-1) = 0$ .  $\text{nums}[0] + \text{nums}[3] + \text{nums}[4] = (-1) + 2 + (-1) = 0$ . 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

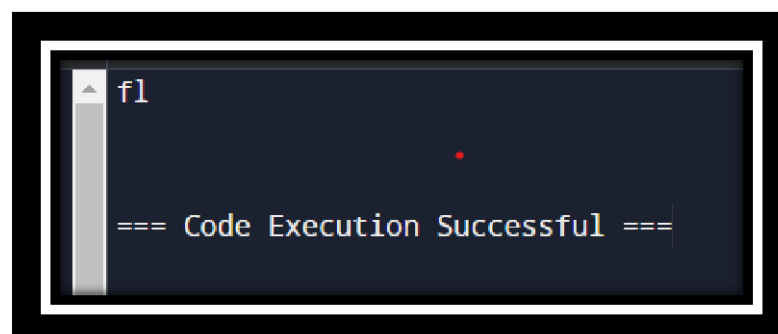
### Program:

```

1 def threeSum(nums):
2     nums.sort()
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
11             if total == 0:
12                 result.append([nums[i], nums[left], nums[right]])
13                 while left < right and nums[left] == nums[left + 1]:
14                     left += 1
15                 while left < right and nums[right] == nums[right - 1]:
16                     right -= 1
17                 left += 1
18                 right -= 1
19             elif total < 0:
20                 left += 1
21             else:
22                 right -= 1
23
24     return result
25

```

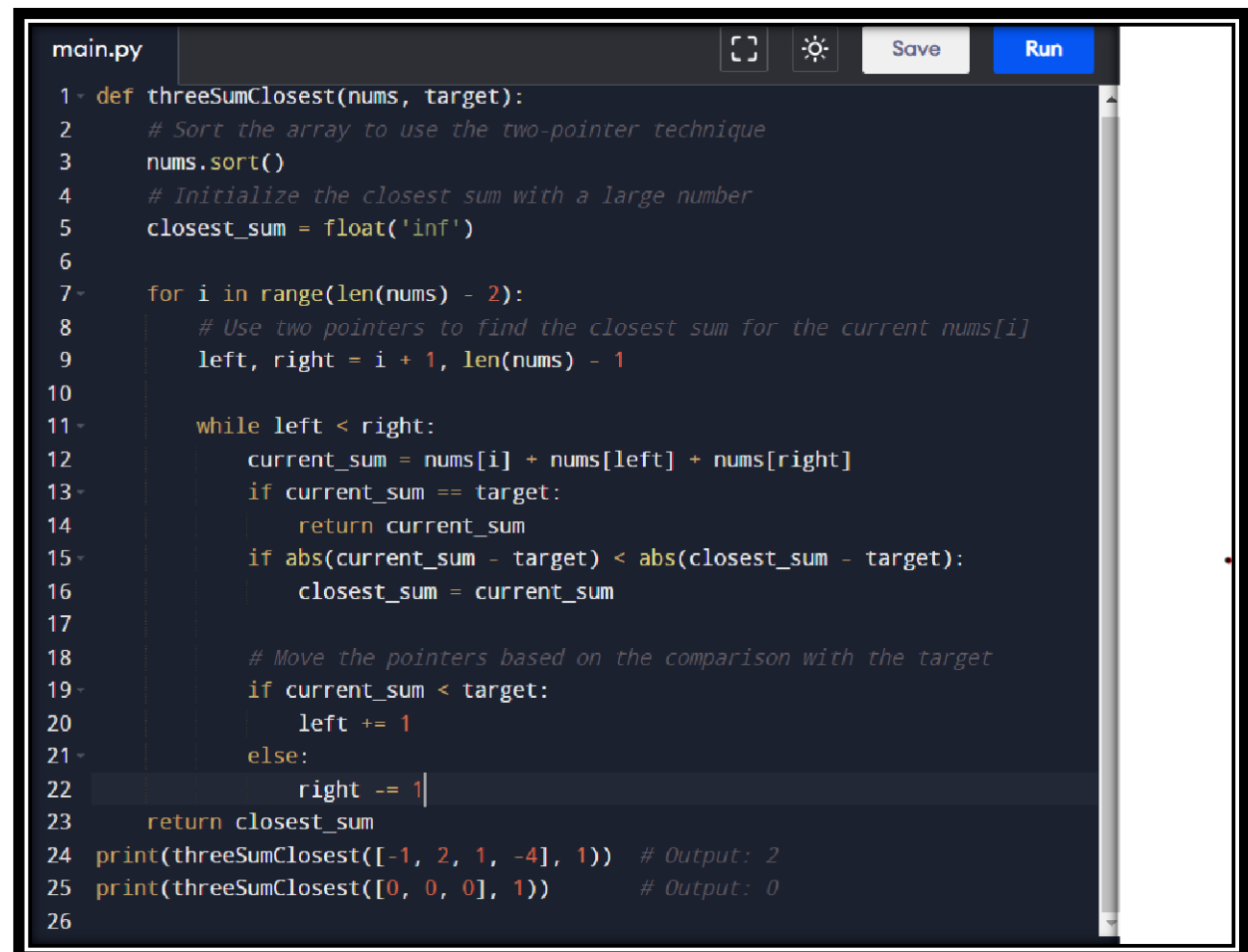
### Output:



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.

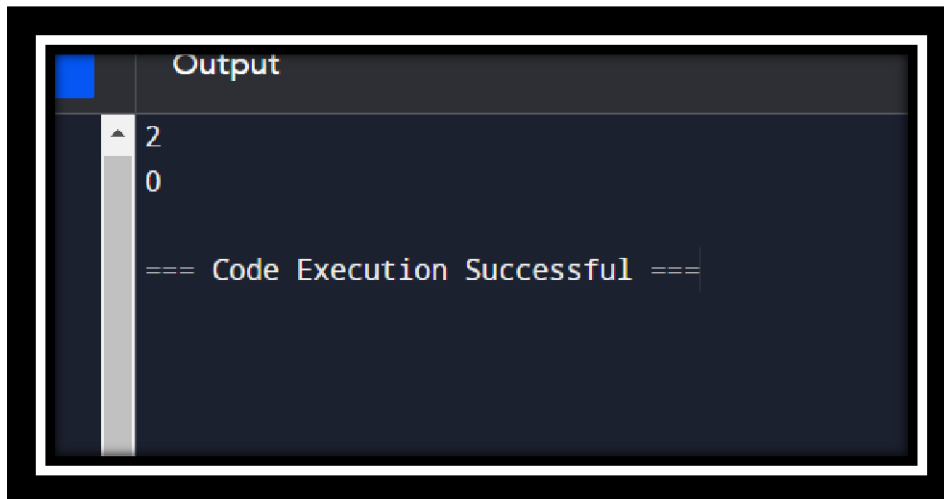
You may assume that each input would have exactly one solution. Example 1: Input: nums = [-1,2,1,-4], target = 1 Output: 2 Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2). Example 2: Input: nums = [0,0,0], target = 1 Output: 0 Explanation: The sum that is closest to the target is 0. (0 + 0 + 0 = 0).

### Program:

A screenshot of a code editor window titled 'main.py'. The editor has a dark background with light-colored text. At the top right, there are icons for a file explorer, a search icon, and buttons labeled 'Save' and 'Run'. The code is a Python function 'threeSumClosest' that takes a list of numbers and a target. It sorts the list and uses a two-pointer technique to find the closest sum of three numbers. The code includes comments in a lighter font. The function returns the closest sum. At the bottom, there are two print statements with comments showing the expected output for the examples provided in the text.

```
main.py
1 def threeSumClosest(nums, target):
2     # Sort the array to use the two-pointer technique
3     nums.sort()
4     # Initialize the closest sum with a large number
5     closest_sum = float('inf')
6
7     for i in range(len(nums) - 2):
8         # Use two pointers to find the closest sum for the current nums[i]
9         left, right = i + 1, len(nums) - 1
10
11         while left < right:
12             current_sum = nums[i] + nums[left] + nums[right]
13             if current_sum == target:
14                 return current_sum
15             if abs(current_sum - target) < abs(closest_sum - target):
16                 closest_sum = current_sum
17
18             # Move the pointers based on the comparison with the target
19             if current_sum < target:
20                 left += 1
21             else:
22                 right -= 1
23     return closest_sum
24 print(threeSumClosest([-1, 2, 1, -4], 1)) # Output: 2
25 print(threeSumClosest([0, 0, 0], 1))    # Output: 0
26
```

Output:

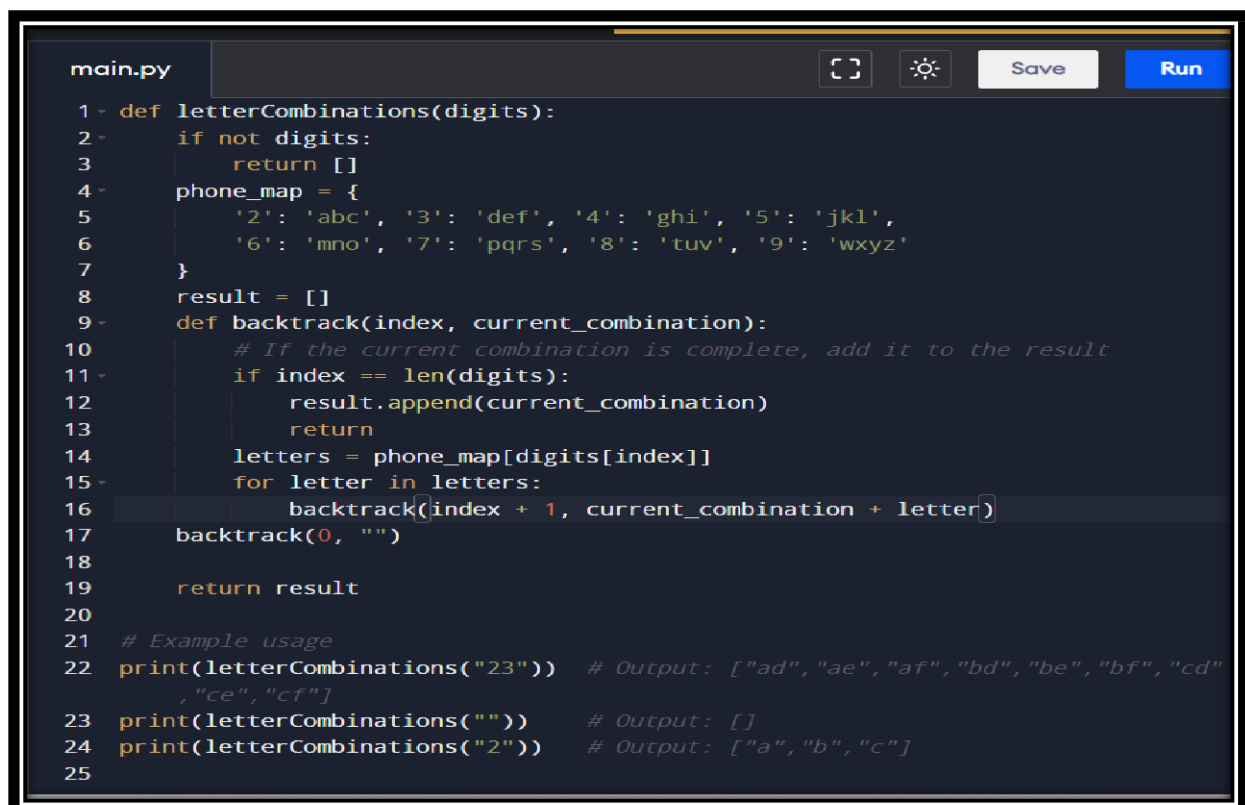


```
Output
2
0

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

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

**Program:**



```
main.py
1- def letterCombinations(digits):
2-     if not digits:
3-         return []
4-     phone_map = {
5-         '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
6-         '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
7-     }
8-     result = []
9-     def backtrack(index, current_combination):
10-         # If the current combination is complete, add it to the result
11-         if index == len(digits):
12-             result.append(current_combination)
13-             return
14-         letters = phone_map[digits[index]]
15-         for letter in letters:
16-             backtrack(index + 1, current_combination + letter)
17-     backtrack(0, "")
18-
19-     return result
20-
21- # Example usage
22- print(letterCombinations("23")) # Output: ["ad", "ae", "af", "bd", "be", "bf", "cd",
23-                                     "ce", "cf"]
24- print(letterCombinations("")) # Output: []
25- print(letterCombinations("2")) # Output: ["a", "b", "c"]
```

**Output:**



```
n      Output
[ 'ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf' ]
[ ]
[ 'a', 'b', 'c' ]

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

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.   
 • `nums[a] + nums[b] + nums[c] + nums[d] == 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):
    # Sort the array to facilitate the two-pointer approach
    nums.sort()
    result = []
    n = len(nums)

    # Outer two loops to fix the first two numbers
    for i in range(n - 3):
        # Skip duplicate elements for the first number
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        for j in range(i + 1, n - 2):
            # Skip duplicate elements for the second number
            if j > i + 1 and nums[j] == nums[j - 1]:
                continue

            # Two-pointer approach for the remaining two numbers
            left, right = j + 1, n - 1
            while left < right:
                current_sum = nums[i] + nums[j] + nums[left] + nums[right]
                if current_sum == target:
                    result.append([nums[i], nums[j], nums[left], nums[right]])

                    # Move left and right pointers to the next unique elements
                    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 current_sum < target:
                    left += 1
                else:
                    right -= 1

            return result

# Example usage
print(fourSum([1, 0, -1, 0, -2, 2], 0)) # Output: [[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
print(fourSum([2, 2, 2, 2, 2], 8))      # Output: [[2, 2, 2, 2]]
```

**Output:**

```
['ad', 'ae', 'af', 'bd', 'be', 'bf', 'cd', 'ce', 'cf']  
[]  
['a', 'b', 'c']
```

```
=== 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]

**Program:**

```
File Edit Format Run Options Windows Help  
class ListNode:  
    def __init__(self, val=0, next=None):  
        self.val = val  
        self.next = next  
  
def removeNthFromEnd(head, n):  
    # Create a dummy node to handle edge cases smoothly  
    dummy = ListNode(0)  
    dummy.next = head  
    first = dummy  
    second = dummy  
  
    # Move the first pointer n+1 steps ahead  
    for _ in range(n + 1):  
        first = first.next  
  
    # Move both first and second pointers until first reaches the end  
    while first is not None:  
        first = first.next  
        second = second.next  
  
    # Remove the nth node from the end  
    second.next = second.next.next  
  
    # Return the new head of the list  
    return dummy.next  
  
# Helper function to create a linked list from a list  
def create_linked_list(lst):  
    if not lst:  
        return None  
    head = ListNode(lst[0])  
    current = head  
    for value in lst[1:]:  
        current.next = ListNode(value)  
        current = current.next  
    return head  
  
# Helper function to convert linked list to a list  
def linked_list_to_list(head):  
    result = []  
    current = head  
    while current:  
        result.append(current.val)  
        current = current.next  
    return result
```

**Output:**

```
Output
[1, 2, 3, 5]
[]
[1]

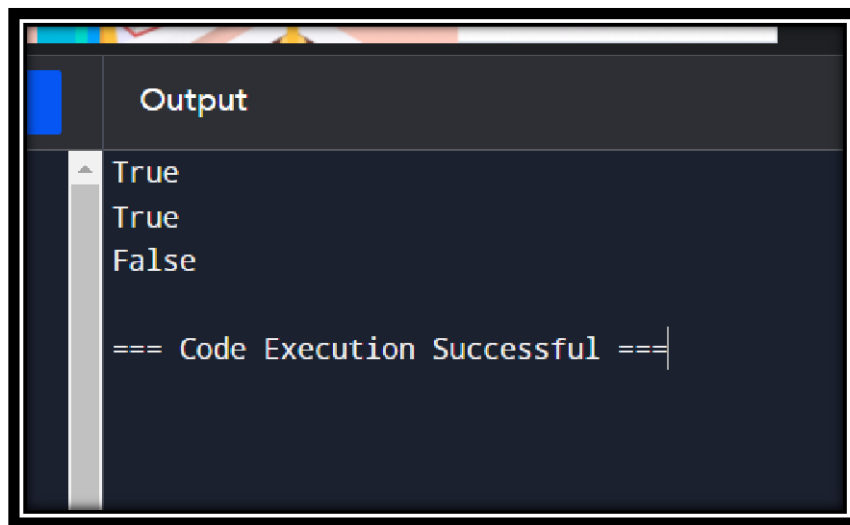
=== Code Execution Successful ===
```

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

Program:

```
1- def isValid(s):
2-     # Dictionary to hold matching pairs of brackets
3-     bracket_map = {'(': ')', '{': '}', '[': ']'}
4-     # Stack to keep track of opening brackets
5-     stack = []
6-
7-     for char in s:
8-         if char in bracket_map:
9-             # Pop the topmost element from the stack if it's not empty,
10-             # otherwise use a dummy value
11-             top_element = stack.pop() if stack else '#'
12-             # Check if the top element matches the corresponding opening
13-             # bracket
14-             if bracket_map[char] != top_element:
15-                 return False
16-         else:
17-             # Push the opening bracket onto the stack
18-             stack.append(char)
19-
20-     # If the stack is empty, all brackets were matched correctly
21-     return not stack
22-
23- # Example usage
24- print(isValid("()"))           # Output: true
25- print(isValid "()[]{}"))      # Output: true
26- print(isValid("("))           # Output: false
```

**Output:**



The image shows a screenshot of a Jupyter Notebook's output area. The output is displayed in a dark-themed window with a title bar that says "Output". The content of the output is as follows:

```
True
True
False

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

The output consists of three lines of text: "True", "True", and "False", which are the results of a loop. Below these, there is a separator line "=== Code Execution Successful ===" indicating that the code executed without errors. A vertical scrollbar is visible on the left side of the output area.