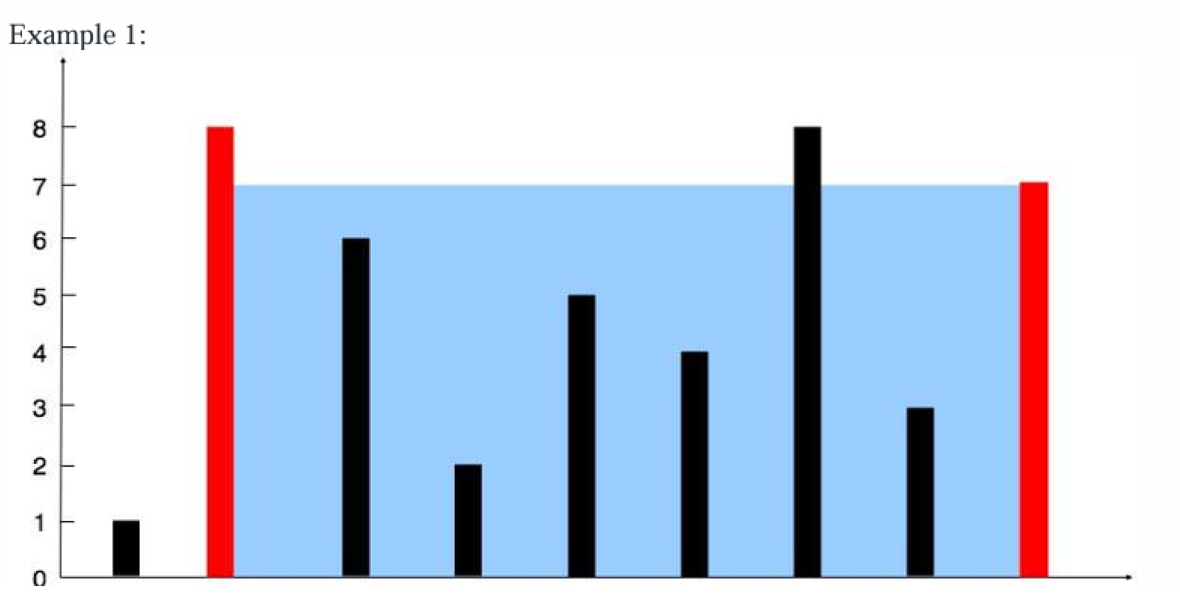
11. Container With Most Water You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]). Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. Notice that you may not slant the container. Example 1:

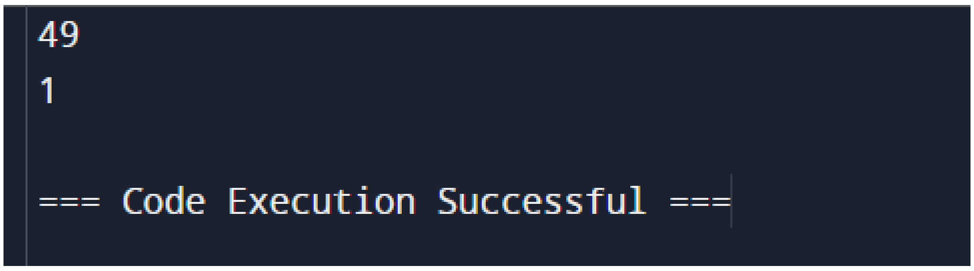


Input: height = [1,8,6,2,5,4,8,3,7] Output: 49 Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49. Example 2: Input: height = [1,1] Output: 1 Constraints: ● n == height.length ● 2 <= n <= 105 ● 0 <= height[i] <= 104

Code:

```
def max_area(height):
    left = 0
    right = len(height) - 1
    max_area = 0

    while left < right:
        width = right - left
        current_height = min(height[left], height[right])
        current_area = width * current_height
        max_area = max(max_area, current_area)

        if height[left] < height[right]:
            left += 1
        else:
            right -= 1

    return max_area

height1 = [1,8,6,2,5,4,8,3,7]
print(max_area(height1))

height2 = [1,1]
print(max_area(height2))
```

output:



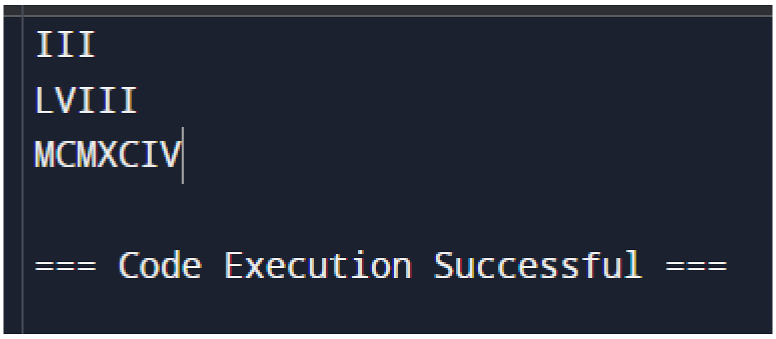
```
49
1

=== 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. Constraints: ● 1 <= num <= 3999

Code:

```
def int_to_roman(num):

    roman_values = [
        (1000, 'M'), (900, 'CM'), (500, 'D'), (400, 'CD'),
        (100, 'C'), (90, 'XC'), (50, 'L'), (40, 'XL'),
        (10, 'X'), (9, 'IX'), (5, 'V'), (4, 'IV'), (1, 'I')
    ]

    roman_numeral = ""

    for value, symbol in roman_values:
        while num >= value:
            roman_numeral += symbol
            num -= value

    return roman_numeral

print(int_to_roman(3))
print(int_to_roman(58))
print(int_to_roman(1994))
```

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

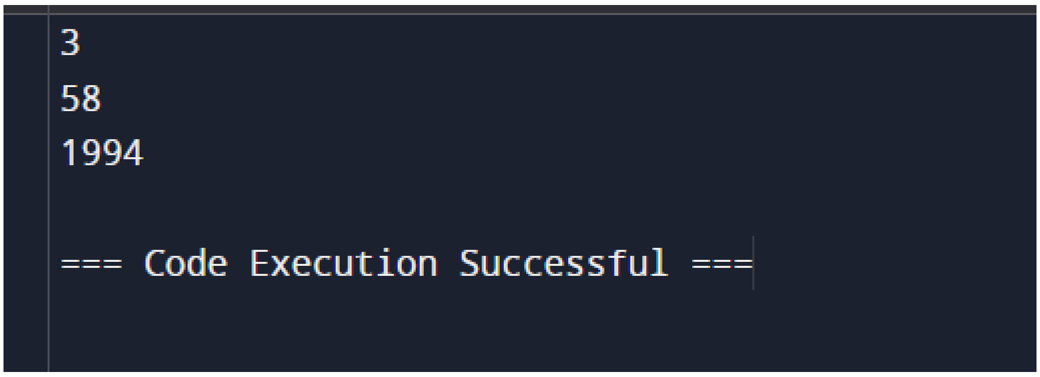
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

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 Explanation: M = 1000, CM = 900, XC = 90 and IV = 4. Constraints: ● 1 <= s.length <= 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].

Code:

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

    total = 0
    n = len(s)

    for i in range(n):

        if i < n - 1 and roman_values[s[i]] < roman_values[s[i + 1]]:
            total -= roman_values[s[i]]
        else:
            total += roman_values[s[i]]

    return total

print(roman_to_int("III"))
print(roman_to_int("LVIII"))
print(roman_to_int("MCMXCIV"))
```

output:



```
3
58
1994

=== 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: ""

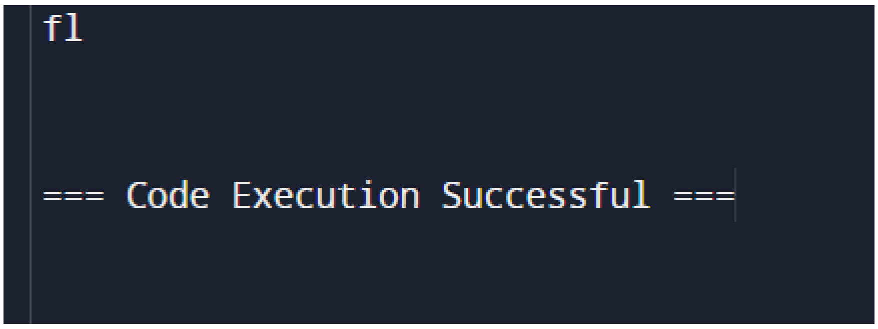
Explanation: There is no common prefix among the input strings.

Constraints:

- 1 <= strs.length <= 200
- 0 <= strs[i].length <= 200
- strs[i] consists of only lowercase English letters

Code:

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

    prefix = strs[0]

    for string in strs[1:]:
        while string[:len(prefix)] != prefix:
            prefix = prefix[:len(prefix) - 1]
            if not prefix:
                return ""

    return prefix

print(longest_common_prefix(["flower", "flow", "flight"]))
print(longest_common_prefix(["dog", "racecar", "car"]))
```



```
fl


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

15. 3Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != 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.

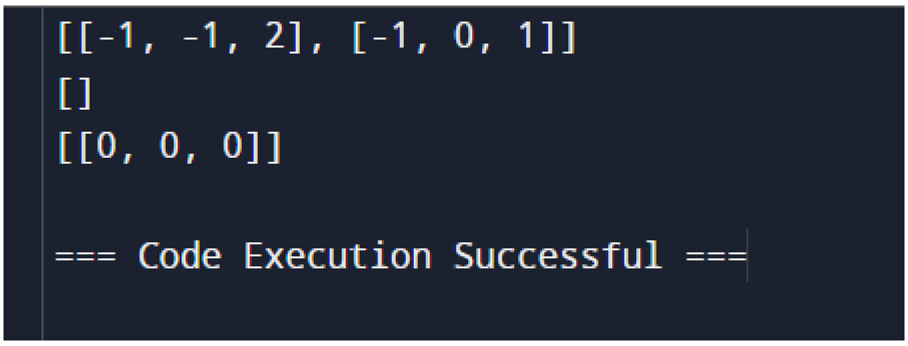
Constraints:

- 3 <= nums.length <= 3000
- -105 <= nums[i] <= 105

Code:

```
def three_sum(nums):
    nums.sort()
    triplets = []
    n = len(nums)

    for i in range(n):
        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]])

                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


print(three_sum([-1, 0, 1, 2, -1, -4]))
print(three_sum([0, 1, 1]))
print(three_sum([0, 0, 0]))
```

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

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

Constraints:

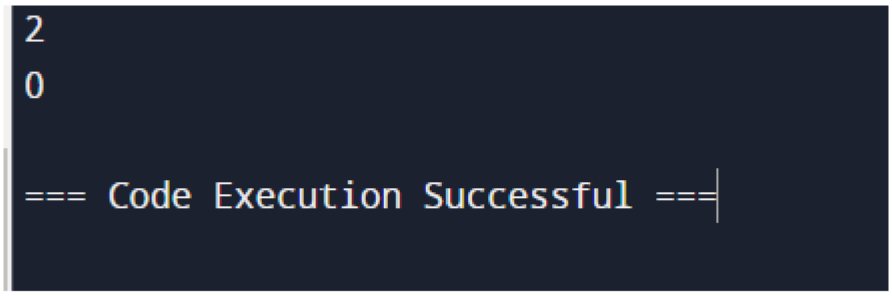
- 3 <= nums.length <= 500
- -1000 <= nums[i] <= 1000
- -104 <= target <= 104

Code:

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

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

        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]



            if abs(current_sum - target) < abs(closest_sum - target):
                closest_sum = current_sum

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

                return current_sum

    return closest_sum

print(three_sum_closest([-1, 2, 1, -4], 1))
```

print(three_sum_closest([0, 0, 0], 1))

ouput:

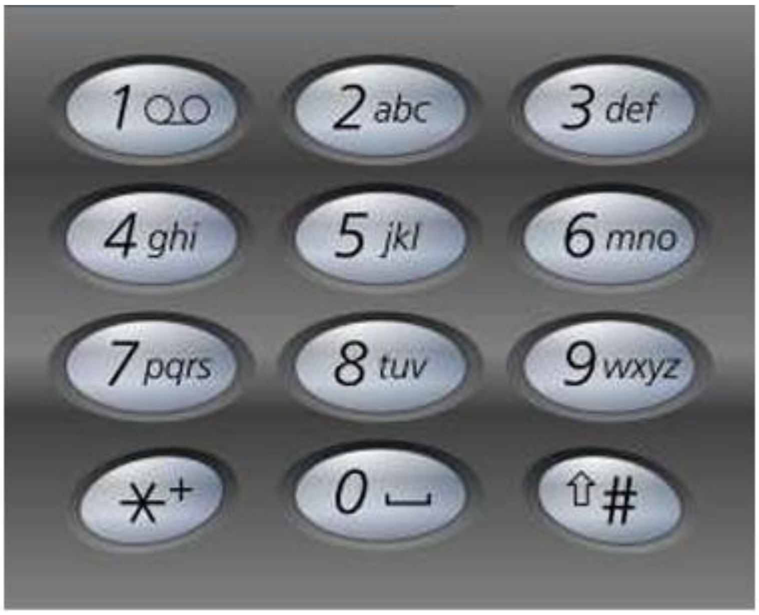


```
2
0

=== 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"]

Constraints:

● 0 <= digits.length <= 4

● digits[i] is a digit in the range ['2', '9'].

Code:

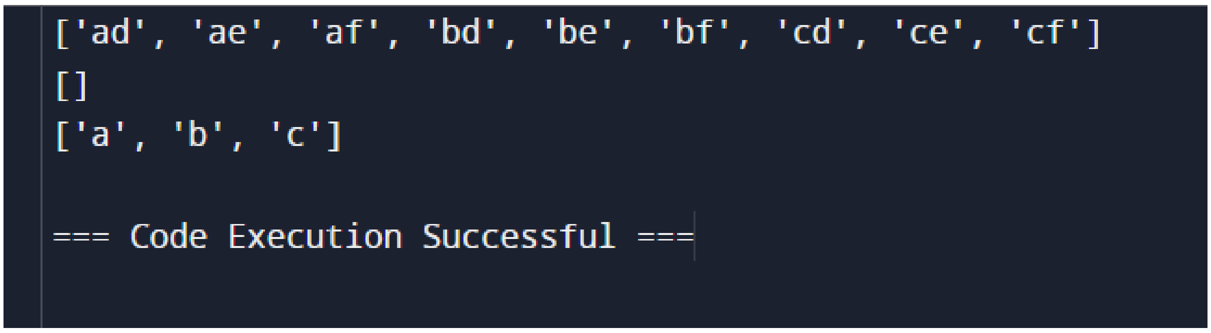
```
def letter_combinations(digits):
    if not digits:
        return []

    phone_map = {
        '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl',
        '6': 'mno', '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
    }

    result = []

    def backtrack(index, path):
        if index == len(digits):
            result.append("".join(path))
            return

        possible_letters = phone_map[digits[index]]

        for letter in possible_letters:
            path.append(letter)
            backtrack(index + 1, path)
            path.pop()
    backtrack(0, [])

    return result
```

print(letter_combinations("23"))

print(letter_combinations(""))

print(letter_combinations("2"))

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 <= 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]]

Constraints:

● 1 <= nums.length <= 200

● -109 <= nums[i] <= 109

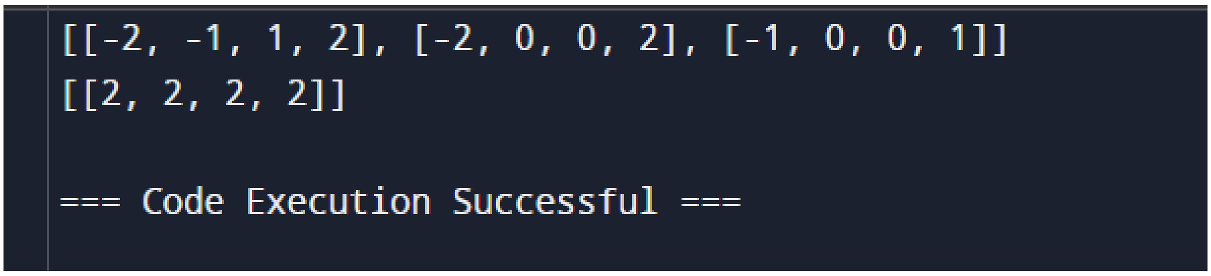
● -109 <= target <= 109

Code:

def four_sum(nums, target):

```
    def k_sum(nums, target, k):
        res = []
        if not nums:
            return res
        average_value = target // k

        if average_value < nums[0] or nums[-1] < average_value:
            return res
        if k == 2:
            return two_sum(nums, target)
        for i in range(len(nums)):
            if i == 0 or nums[i - 1] != nums[i]:
                for subset in k_sum(nums[i + 1:], target - nums[i], k - 1):
                    res.append([nums[i]] + subset)
        return res

    def two_sum(nums, target):
        res = []
        s = set()
        for i in range(len(nums)):
            if len(res) == 0 or res[-1][1] != nums[i]:
                if target - nums[i] in s:
                    res.append([target - nums[i], nums[i]])
            s.add(nums[i])
        return res

    nums.sort()
    return k_sum(nums, target, 4)

print(four_sum([1, 0, -1, 0, -2, 2], 0))
print(four_sum([2, 2, 2, 2, 2], 8))
```

output:

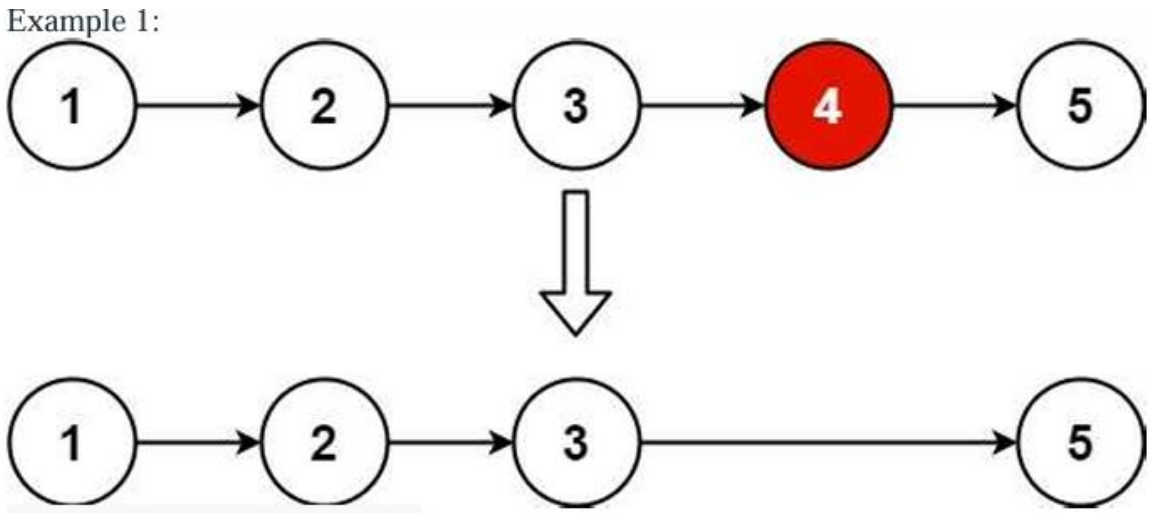


```
[[-2, -1, 1, 2], [-2, 0, 0, 2], [-1, 0, 0, 1]]
[[2, 2, 2, 2]]

=== 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 <= sz <= 30
- 0 <= Node.val <= 100

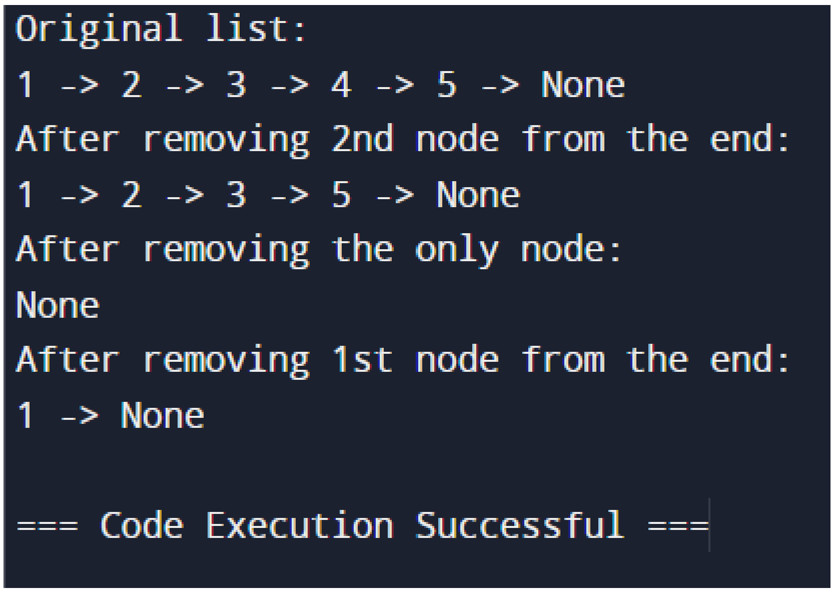
● 1 <= n <= sz

Code:

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

def remove_nth_from_end(head, n):
    dummy = ListNode(0)
    dummy.next = head
    first = dummy
    second = dummy

    for _ in range(n + 1):
        second = second.next

    while second:
        first = first.next
        second = second.next

    first.next = first.next.next

    return dummy.next

def print_list(head):
    while head:
        print(head.val, end=" -> ")
        head = head.next
    print("None")

head = ListNode(1)
```

```
head.next = ListNode(2)

head.next.next = ListNode(3)

head.next.next.next = ListNode(4)

head.next.next.next.next = ListNode(5)


print("Original list:")

print_list(head)

new_head = remove_nth_from_end(head, 2)

print("After removing 2nd node from the end:")

print_list(new_head)


head = ListNode(1)

new_head = remove_nth_from_end(head, 1)

print("After removing the only node:")

print_list(new_head)


head = ListNode(1)

head.next = ListNode(2)

new_head = remove_nth_from_end(head, 1)

print("After removing 1st node from the end:")

print_list(new_head)
```

output:



```
Original list:
1 -> 2 -> 3 -> 4 -> 5 -> None
After removing 2nd node from the end:
1 -> 2 -> 3 -> 5 -> None
After removing the only node:
None
After removing 1st node from the end:
1 -> None

=== 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

Constraints:

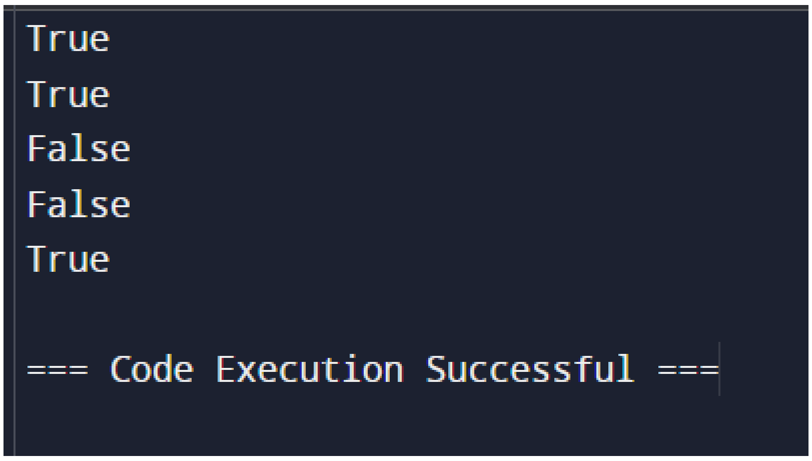
● 1 <= s.length <= 104

● s consists of parentheses only '()[]{}'.

Code:

```
def is_valid_parentheses(s):
    bracket_map = {')': '(', ']': '[', '}': '{'}
    stack = []

    for char in s:
        if char in bracket_map:

            top_element = stack.pop() if stack else '#'

            if bracket_map[char] != top_element:
                return False
        else:
            stack.append(char)

    return not stack

print(is_valid_parentheses("()"))
print(is_valid_parentheses("()[]{}"))
print(is_valid_parentheses("(]"))
print(is_valid_parentheses("([)]"))
print(is_valid_parentheses("{[]}"))
```

output:



```
True
True
False
False
True

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