

ASSIGNMENT-5

NAME:K.R.VISHNU CHAITHANYA

REGT.NO:192372057

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]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`

Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0,1]`

Constraints:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- Only one valid answer exists.

Code:

```

def twoSum(nums, target):
    num_to_index = {}
    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

print(twoSum([2,7,11,15], 9))
print(twoSum([3,2,4], 6))
print(twoSum([3,3], 6))

```

Output:

```

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

=== Code Execution Successful ===

```

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.

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]

Constraints:

- The number of nodes in each linked list is in the range [1, 100].
- $0 \leq \text{Node.val} \leq 9$
- It is guaranteed that the list represents a number that does not have leading zeros.

Code:

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

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

    while l1 or l2 or carry:
        val1 = (l1.val if l1 else 0)
        val2 = (l2.val if l2 else 0)
        carry, out = divmod(val1 + val2 + carry, 10)

        current.next = ListNode(out)
        current = current.next

        l1 = (l1.next if l1 else None)
        l2 = (l2.next if l2 else None)

    return dummy.next

def create_linked_list(lst):
    dummy = ListNode()
    current = dummy
```

```

        current.next = ListNode(out)
        current = current.next

    l1 = (l1.next if l1 else None)
    l2 = (l2.next if l2 else None)

    return dummy.next

def create_linked_list(lst):
    dummy = ListNode()
    current = dummy
    for val in lst:
        current.next = ListNode(val)
        current = current.next
    return dummy.next

l1 = create_linked_list([2,4,3])
l2 = create_linked_list([5,6,4])
result = addTwoNumbers(l1, l2)
res_list = []
while result:
    res_list.append(result.val)
    result = result.next
print(res_list)

```

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: $s = \text{"abcabcbb"}$

Output: 3

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

Example 2:

Input: $s = \text{"bbbbbb"}$

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.

Constraints:

- $0 \leq s.length \leq 5 * 10^4$
- s consists of English letters, digits, symbols and spaces.

Code:

```
def lengthOfLongestSubstring(s):  
    char_map = {}  
    left = 0  
    max_length = 0  
  
    for right, char in enumerate(s):  
        if char in char_map and char_map[char] >= left:  
            left = char_map[char] + 1  
        char_map[char] = right  
        max_length = max(max_length, right - left + 1)  
  
    return max_length  
  
print(lengthOfLongestSubstring("abcabcbb"))  
print(lengthOfLongestSubstring("bbbbb"))  
print(lengthOfLongestSubstring("pwwkew"))
```

Output:

Output
3
1
3
=== Code Execution Successful ===

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 $(2 + 3) / 2 = 2.5$.

Constraints:

- `nums1.length == m`
- `nums2.length == n`
- $0 \leq m \leq 1000$
- $0 \leq n \leq 1000$
- $1 \leq m + n \leq 2000$
- $-106 \leq \text{nums1}[i], \text{nums2}[i] \leq 106$

Code:

```
main.py  [Icons] [Save] [Run]
1 def findMedianSortedArrays(nums1, nums2):
2     def find_kth_smallest(a, b, k):
3         if not a:
4             return b[k]
5         if not b:
6             return a[k]
7         mid_a, mid_b = len(a) // 2, len(b) // 2
8         mid_a_val, mid_b_val = a[mid_a], b[mid_b]
9
10        if mid_a + mid_b < k:
11            if mid_a_val > mid_b_val:
12                return find_kth_smallest(a, b[mid_b + 1:], k - mid_b - 1)
13            else:
14                return find_kth_smallest(a[mid_a + 1:], b, k - mid_a - 1)
15        else:
16            if mid_a_val > mid_b_val:
17                return find_kth_smallest(a[:mid_a], b, k)
18            else:
19                return find_kth_smallest(a, b[:mid_b], k)
20
21    total_len = len(nums1) + len(nums2)
22    if total_len % 2 == 1:
23        return find_kth_smallest(nums1, nums2, total_len // 2)
24    else:
25        return (find_kth_smallest(nums1, nums2, total_len // 2 - 1) +
                find_kth_smallest(nums1, nums2, total_len // 2)) / 2
```

Output:

```
Output
3
1
3
=== Code Execution Successful ===
```

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"

Constraints:

- $1 \leq s.length \leq 1000$
- s consist of only digits and English letters.

```
main.py
1 def longestPalindrome(s):
2     def expand_around_center(left, right):
3         while left >= 0 and right < len(s) and s[left] == s[right]:
4             left -= 1
5             right += 1
6         return s[left + 1:right]
7
8     longest = ""
9     for i in range(len(s)):
10        odd_palindrome = expand_around_center(i, i)
11        even_palindrome = expand_around_center(i, i + 1)
12        longest = max(longest, odd_palindrome, even_palindrome, key=len)
13
14    return longest
15
16 print(longestPalindrome("babad"))
17 print(longestPalindrome("cbbd"))
18
```

Ouput:

```
Output
bab
bb

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

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
```

Example 3:

Input: s = "A", numRows = 1

Output: "A"

Constraints:

- $1 \leq s.length \leq 1000$
- s consists of English letters (lower-case and upper-case), ',' and '.'.
- $1 \leq numRows \leq 1000$

Code:

```

1 def convert(s, numRows):
2     if numRows == 1:
3         return s
4
5     rows = [''] * numRows
6     cur_row = 0
7     going_down = False
8
9     for char in s:
10        rows[cur_row] += char
11        if cur_row == 0 or cur_row == numRows - 1:
12            going_down = not going_down
13            cur_row += 1 if going_down else -1
14
15    return ''.join(rows)
16
17 print(convert("PAYPALISHIRING", 3))
18 print(convert("PAYPALISHIRING", 4))
19 print(convert("A", 1))
20

```

Output:

Output
PAHNAPLSIIGYIR
PINALSIGYAHRPI
A
=== Code Execution Successful ===

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 $[-2^{31}, 2^{31} - 1]$, 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

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$

Code:

```
main.py
1 def reverse(x):
2     sign = -1 if x < 0 else 1
3     x *= sign
4     reversed_x = int(str(x)[::-1])
5     if reversed_x > 2**31 - 1:
6         return 0
7     return sign * reversed_x
8
9 print(reverse(123))
10 print(reverse(-123))
11 print(reverse(120))
12
```

Ouput:

```
Output
321
-321
21

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

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 $[-2^{31}, 2^{31} - 1]$, then clamp the integer so that it remains in the range. Specifically, integers less than -2^{31} should be clamped to -2^{31} , and integers greater than $2^{31} - 1$ should be clamped to $2^{31} - 1$.
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 $[-231, 231 - 1]$, 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 ' '.

code:

```

s = s.strip()
if not s:
    return 0

sign = 1
start = 0
if s[0] in ('-', '+'):
    if s[0] == '-':
        sign = -1
    start = 1

result = 0
for char in s[start:]:
    if not char.isdigit():
        break
    result = result * 10 + int(char)
    if sign * result > 2**31 - 1:
        return 2**31 - 1
    if sign * result < -2**31:
        return -2**31

return sign * result

print(myAtoi("42"))
print(myAtoi("   -42"))
print(myAtoi("4193 with words"))

```

Output:


```
Output
42
-42
4193

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

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:

- $-2^{31} \leq x \leq 2^{31} - 1$

Code:

```

main.py
1 def isPalindrome(x):
2     if x < 0:
3         return False
4     return str(x) == str(x)[::-1]
5
6 print(isPalindrome(121))
7 print(isPalindrome(-121))
8 print(isPalindrome(10))
9

```

Output:

```

Output
True
False
False

=== Code Execution Successful ===

```

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.

Code:

```

1 def isMatch(s, p):
2     dp = [[False] * (len(p) + 1) for _ in range(len(s) + 1)]
3     dp[0][0] = True
4
5     for j in range(2, len(p) + 1):
6         if p[j - 1] == '*':
7             dp[0][j] = dp[0][j - 2]
8
9     for i in range(1, len(s) + 1):
0         for j in range(1, len(p) + 1):
1             if p[j - 1] == '.' or p[j - 1] == s[i - 1]:
2                 dp[i][j] = dp[i - 1][j - 1]
3             elif p[j - 1] == '*':
4                 dp[i][j] = dp[i][j - 2] or (dp[i - 1][j] if p[j - 2] == s[i - 1] or
5                                         p[j - 2] == '.' else False)
6
7     return dp[-1][-1]
8
9 print(isMatch("aa", "a"))
10 print(isMatch("aa", "a*"))
11 print(isMatch

```

Output:

Output
True
False
False
=== Code Execution Successful ===

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:

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

Code:

```
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         max_area = max(max_area, min(height[left], height[right]) * width)
8         if height[left] < height[right]:
9             left += 1
10        else:
11            right -= 1
12
13    return max_area
14
15
16 print(maxArea([1,8,6,2,5,4,8,3,7]))
17 print(maxArea([1,1]))
18
```

```
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:

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

Code:

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    roman_num = ''
15    for i in range(len(val)):
16        while num >= val[i]:
17            num -= val[i]
18            roman_num += syms[i]
19    return roman_num
20
21 print(intToRoman(3))
22 print(intToRoman(58))
23 print(intToRoman(1994))
24
```

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

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].

Code:

```
main.py
1 def romanToInt(s):
2     roman_to_int = {
3         'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100,
4         'D': 500, 'M': 1000
5     }
6     total = 0
7     prev_value = 0
8     for char in reversed(s):
9         value = roman_to_int[char]
10        if value < prev_value:
11            total -= value
12        else:
13            total += value
14            prev_value = value
15    return total
16
17 print(romanToInt("III"))
18 print(romanToInt("LVIII"))
19 print(romanToInt("MCMXCIV"))
20
```

Output:

```
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:

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

Code:

```
main.py
1 def longestCommonPrefix(strs):
2     if not strs:
3         return ""
4     prefix = strs[0]
5     for string in strs[1:]:
6         while string[:len(prefix)] != prefix and prefix:
7             prefix = prefix[:-1]
8         if not prefix:
9             break
10    return prefix
11
12 print(longestCommonPrefix(["flower","flow","flight"]))
13 print(longestCommonPrefix(["dog","racecar","car"]))
```

Output:

```
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:

$nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0$.

$nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0$.

$nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0$.

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:

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

Code:

```
main.py  [ ] [ ] Save Run
2     nums.sort()
3     result = []
4     for i in range(len(nums)):
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 print(threeSum([-1,0,1,2,-1,-4]))
25 print(threeSum([0,1,1]))
26 print(threeSum([0,0,0]))
27
```

Output:

```
Output
^ f1
=== 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:

- $3 \leq \text{nums.length} \leq 500$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^4 \leq \text{target} \leq 10^4$

Code:

```
main.py  [ ] [ ] Save Run

1 def threeSumClosest(nums, target):
2     nums.sort()
3     closest_sum = float('inf')
4     for i in range(len(nums)):
5         left, right = i + 1, len(nums) - 1
6         while left < right:
7             current_sum = nums[i] + nums[left] + nums[right]
8             if abs(current_sum - target) < abs(closest_sum - target):
9                 closest_sum = current_sum
10            if current_sum < target:
11                left += 1
12            elif current_sum > target:
13                right -= 1
14            else:
15                return current_sum
16    return closest_sum
17
18 print(threeSumClosest([-1,2,1,-4], 1))
19 print(threeSumClosest([0,0,0], 1))
20
```

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"]

Constraints:

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

Code:


```
main.py  [ ] [ ] Save Run

1 def letterCombinations(digits):
2     if not digits:
3         return []
4     phone = {
5         '2': 'abc', '3': 'def', '4': 'ghi', '5': 'jkl', '6': 'mno',
6         '7': 'pqrs', '8': 'tuv', '9': 'wxyz'
7     }
8     result = ['']
9     for digit in digits:
10        temp = []
11        for combo in result:
12            for letter in phone[digit]:
13                temp.append(combo + letter)
14        result = temp
15    return result
16
17 print(letterCombinations("23"))
18 print(letterCombinations(""))
19 print(letterCombinations("2")) #
20
```

Output:

```
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]]

Constraints:

- $1 \leq \text{nums.length} \leq 200$
- $-109 \leq \text{nums}[i] \leq 109$
- $-109 \leq \text{target} \leq 109$

Code:

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

Output:

```
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]

Constraints:

- The number of nodes in the list is sz.
- $1 \leq sz \leq 30$
- $0 \leq \text{Node.val} \leq 100$
- $1 \leq n \leq sz$

Code:

```
main.py   Save Run

1 class ListNode:
2     def __init__(self, val=0, next=None):
3         self.val = val
4         self.next = next
5 def removeNthFromEnd(head, n):
6     dummy = ListNode(0)
7     dummy.next = head
8     first = dummy
9     second = dummy
10    for _ in range(n + 1):
11        first = first.next
12    while first:
13        first = first.next
14        second = second.next
15    second.next = second.next.next
16    return dummy.next
17 def create_linked_list(lst):
18     dummy = ListNode()
19     current = dummy
20    for val in lst:
21        current.next = ListNode(val)
22        current = current.next
23    return dummy.next
24 head = create_linked_list([1,2,3,4,5])
25 n = 2
26 result = removeNthFromEnd(head, n)
```

Output:

```
Output

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

=== 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:

- $1 \leq s.length \leq 104$
- s consists of parentheses only '()[]{}'.

Code:

```
main.py  [Icons] Sa

1 def isValid(s):
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 print(isValid("()"))
16 print(isValid("()[{}]")
17 print(isValid("]"))
```

Output:

```
Output

True
True
False

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