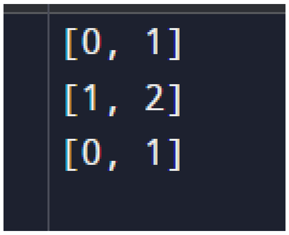
KONGARA SAI
192365025
DESIGN ANALYSIS OF ALGORITHM
CSA0674
ASSIGNMENT-1

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 <= nums.length <= 104 ● -109 <= nums[i] <= 109 ● -109 <= target <= 109 ● Only one valid answer exists.

PROGRAM :

```
def two_sum(nums, target):
    num_to_index = {}

    for index, num in enumerate(nums):
        complement = target - num

        if complement in num_to_index:
            return [num_to_index[complement], index]

        num_to_index[num] = index

    raise ValueError("No two sum solution")

# Test the function with the provided examples
print(two_sum([2, 7, 11, 15], 9))  # Output: [0, 1]
print(two_sum([3, 2, 4], 6))       # Output: [1, 2]
print(two_sum([3, 3], 6))          # Output: [0, 1]
```



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

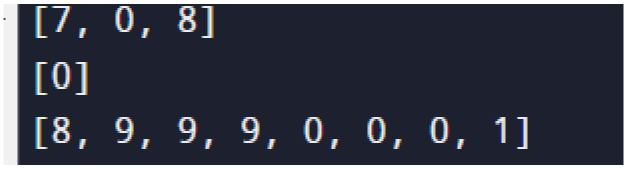
1. Add Two Numbers You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list. You may assume the two numbers do not contain any leading 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 <= Node.val <= 9 ● It is guaranteed that the list represents a number that does not have leading zeros.

PROGRAM:

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


def add_two_numbers(l1, l2):
    # Initialize a dummy head to simplify the code and a pointer for the result list
    dummy_head = ListNode(0)
    current = dummy_head
    carry = 0

    # Loop until both lists are exhausted and there is no carry left
    while l1 or l2 or carry:
        # Sum the values of the nodes (or 0 if the list is exhausted) and the carry
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0
        total = val1 + val2 + carry

        # Update the carry for the next addition
        carry = total // 10
        current.next = ListNode(total % 10)

        # Move to the next node in the result list
        current = current.next
        # Advance the input lists if possible
        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    # Return the next node of dummy head, which is the start of the result list
    return dummy_head.next

# Helper function to create a linked list from a list of integers
def create_linked_list(numbers):
    dummy_head = ListNode(0)
    current = dummy_head
    for number in numbers:
        current.next = ListNode(number)
        current = current.next
    return dummy_head.next

# Helper function to convert a linked list to a list of integers
def linked_list_to_list(node):
    result = []
    while node:
        result.append(node.val)
        node = node.next
    return result

# Test the function with the provided examples
l1 = create_linked_list([2, 4, 3])
l2 = create_linked_list([5, 6, 4])
```

```
result = add_two_numbers(l1, l2)

print(linked_list_to_list(result))  # Output: [7, 0, 8]


l1 = create_linked_list([0])

l2 = create_linked_list([0])

result = add_two_numbers(l1, l2)

print(linked_list_to_list(result))  # Output: [0]


l1 = create_linked_list([9, 9, 9, 9, 9, 9, 9])

l2 = create_linked_list([9, 9, 9, 9])

result = add_two_numbers(l1, l2)

print(linked_list_to_list(result))  # Output: [8, 9, 9, 9, 0, 0, 0, 1]
```

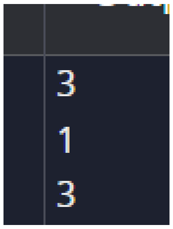


```
[7, 0, 8]
[0]
[8, 9, 9, 9, 0, 0, 0, 1]
```

2. 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. Example 2: Input: s = "bbbbb" Output: 1 Explanation: The answer is "b", with the length of 1. Example 3: Input: s = "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 <= s.length <= 5 * 104 ● s consists of English letters, digits, symbols and spaces.

PROGRAM:

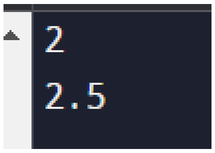
```
def length_of_longest_substring(s):

    char_index = {}

    longest = 0

    start = 0


    for i, char in enumerate(s):

        if char in char_index and char_index[char] >= start:

            start = char_index[char] + 1
        char_index[char] = i
        longest = max(longest, i - start + 1)

    return longest

# Test the function with the provided examples
print(length_of_longest_substring("abcabcbb"))  # Output: 3
print(length_of_longest_substring("bbbbb"))     # Output: 1
print(length_of_longest_substring("pwwkew"))    # Output: 3
```



```
3
1
3
```

3. 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 <= m <= 1000 ● 0 <= n <= 1000 ● 1 <= m + n <= 2000 ● -106 <= nums1[i], nums2[i] <= 106

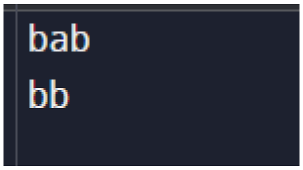
```
def find_median_sorted_arrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    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]:
            # i is too small, must increase it
            imin = i + 1
        elif i > 0 and nums1[i - 1] > nums2[j]:
            # i is too big, must decrease it
            imax = i - 1
        else:
            # i is perfect
            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

# Test the function with the provided examples
print(find_median_sorted_arrays([1, 3], [2]))      # Output: 2.0
print(find_median_sorted_arrays([1, 2], [3, 4]))   # Output: 2.5

def longest_palindromic_substring(s):
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1

    if len(s) == 0:
        return ""

    start, end = 0, 0
    for i in range(len(s)):
        # Expand around the center for odd length palindromes
        left1, right1 = expand_around_center(i, i)
        # Expand around the center for even length palindromes
        left2, right2 = expand_around_center(i, i + 1)

        if right1 - left1 > end - start:
            start, end = left1, right1
        if right2 - left2 > end - start:
            start, end = left2, right2

    return s[start:end + 1]

# Test the function with the provided examples
print(longest_palindromic_substring("babad"))  # Output: "bab" or "aba"
print(longest_palindromic_substring("cbbd"))   # Output: "bb"
```



```
2
2.5
```

4. 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 <= s.length <= 1000 ● s consist of only digits and English letters.

```
def longest_palindromic_substring(s):
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return left + 1, right - 1

    if len(s) == 0:
        return ""

    start, end = 0, 0
    for i in range(len(s)):
         left1, right1 = expand_around_center(i, i)
         left2, right2 = expand_around_center(i, i + 1)

        if right1 - left1 > end - start:
            start, end = left1, right1
        if right2 - left2 > end - start:
            start, end = left2, right2

    return s[start:end + 1]


print(longest_palindromic_substring("babad"))  # Output: "bab" or "aba"
print(longest_palindromic_substring("cbbd"))   # Output: "bb"
```

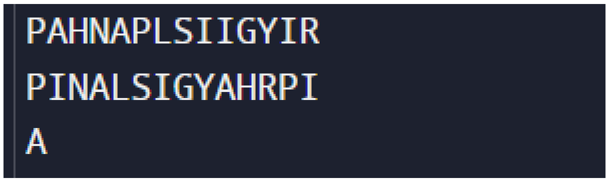


```
bab
bb
```

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 <= s.length <= 1000 ● s consists of English letters (lower-case and upper-case), ',' and '.'. ● 1 <= numRows <= 1000

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

    rows = ['' for _ in range(numRows)]
    current_row = 0
    going_down = False

    for char in s:
        rows[current_row] += char
        if current_row == 0 or current_row == numRows - 1:
            going_down = not going_down
        current_row += 1 if going_down else -1

    return ''.join(rows)

print(convert("PAYPALISHIRING", 3))  # Output: "PAHNAPLSIIGYIR"
print(convert("PAYPALISHIRING", 4))  # Output: "PINALSIGYAHRPI"
print(convert("A", 1))              # Output: "A"
```

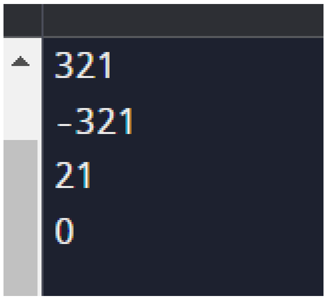


```
PAHNAPLSIIGYIR
PINALSIGYAHRPI
A
```

7. Reverse Integer Given a signed 32-bit integer x, return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0. Assume the

environment does not allow you to store 64-bit integers (signed or unsigned). Example 1: Input: x = 123 Output: 321 Example 2: Input: x = -123 Output: -321 Example 3: Input: x = 120 Output: 21 Constraints: ● -231 <= x <= 231 - 1

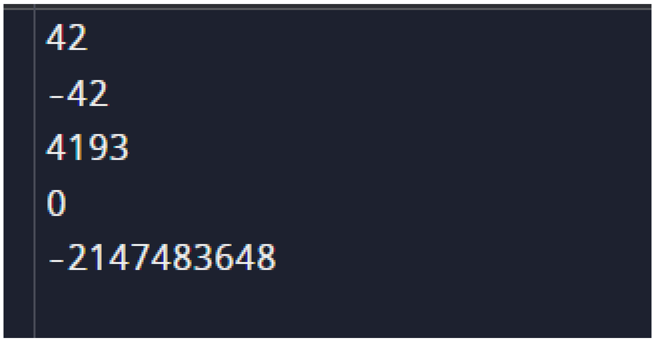
```
def reverse(x):
     INT_MIN, INT_MAX = -2**31, 2**31 - 1

     result = 0
    sign = -1 if x < 0 else 1
    x = abs(x)

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

         if result > (INT_MAX - pop) // 10:
            return 0

        result = result * 10 + pop

    return result * sign

# Test the function with the provided examples
print(reverse(123))   # Output: 321
print(reverse(-123))  # Output: -321
print(reverse(120))   # Output: 21
print(reverse(0))     # Output: 0
```



```
321
-321
21
0
```

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 [-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 <= s.length <= 200 ● s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

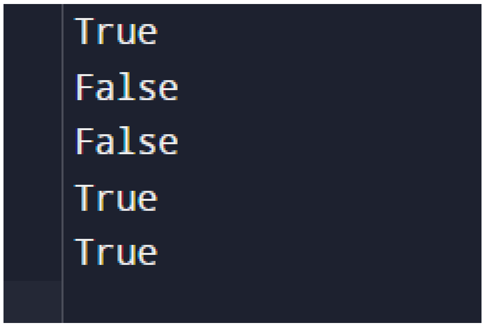
```
def myAtoi(s: str) -> int:
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    i = 0
    n = len(s)

    while i < n and s[i] == ' ':
        i += 1

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

    result = 0
    while i < n and s[i].isdigit():
        digit = int(s[i])

        if result > (INT_MAX - digit) // 10:
            return INT_MIN if sign == -1 else INT_MAX

        result = result * 10 + digit
        i += 1

    return sign * result

print(myAtoi("42"))           # Output: 42
print(myAtoi("   -42"))       # Output: -42
print(myAtoi("4193 with words"))  # Output: 4193
print(myAtoi("words and 987"))    # Output: 0
print(myAtoi("-91283472332"))     # Output: -2147483648 (clamped)
```



```
42
-42
4193
0
-2147483648
```

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 <= x <= 231 – 1

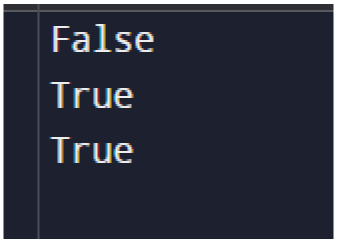
```
def isPalindrome(x: int) -> bool:
    if x < 0:
        return False

    if x != 0 and x % 10 == 0:
        return False

    reversed_half = 0
    while x > reversed_half:
        reversed_half = reversed_half * 10 + x % 10
        x //= 10

    return x == reversed_half or x == reversed_half // 10


# Test cases
print(isPalindrome(121))   # Output: True
print(isPalindrome(-121))  # Output: False
print(isPalindrome(10))    # Output: False
print(isPalindrome(0))     # Output: True
print(isPalindrome(12321)) # Output: True
```



```
True
False
False
True
True
```

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 <= s.length <= 20 ● 1 <= p.length <= 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

```
def isMatch(s: str, p: str) -> bool:
    dp = [[False] * (len(p) + 1) for _ in range(len(s) + 1)]
    dp[0][0] = True


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


    for i in range(1, len(s) + 1):
      for j in range(1, len(p) + 1):
          if p[j - 1] == '*':
              dp[i][j] = dp[i][j - 2] or (dp[i - 1][j] and (s[i - 1] == p[j - 2] or p[j - 2] == '.'))
          elif p[j - 1] == '.' or s[i - 1] == p[j - 1]:
              dp[i][j] = dp[i - 1][j - 1]
    return dp[len(s)][len(p)]


 print(isMatch("aa", "a"))    # Output: False

print(isMatch("aa", "a*"))   # Output: True

print(isMatch("ab", ".*"))   # Output: True
```



```
False
True
True
```