

ASSIGNMENT -1

1. 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` •

Program:

```
def twoSum(nums, target):
```

```
    num_indices = {}
```

```
    for i, num in enumerate(nums):
```

```
        complement = target - num
```

```
        if complement in num_indices:
```

```
            return [num_indices[complement], i]
```

```
    num_indices[num] = i
```

```
nums1 = [2, 7, 11, 15]
```

```
target1 = 9
```

```
print(twoSum(nums1, target1))
```

```
nums2 = [3, 2, 4]
```

```
target2 = 6
```

```
print(twoSum(nums2, target2))
```

```
nums3 = [3, 3]
```

```
target3 = 6
```

```
print(twoSum(nums3, target3))
```

Output:

```
answers are:
```

```
[0, 1]
```

```
[1, 2]
```

```
[0, 1]
```

2. 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]

Program:

```
class ListNode:
```

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

```
        self.val = val
```

```
        self.next = next
```

```
def addTwoNumbers(l1, l2):
```

```
    dummy_head = ListNode()
```

```
    current = dummy_head
```

```
    carry = 0
```

```
    while l1 or l2 or carry:
```

```
        sum_val = carry
```

```
        if l1:
```

```
            sum_val += l1.val
```

```
            l1 = l1.next
```

```
        if l2:
```

```
            sum_val += l2.val
```

```
            l2 = l2.next
```

```
        carry, digit = divmod(sum_val, 10)
```

```
        current.next = ListNode(digit)
```

```
        current = current.next
```

```

    return dummy_head.next
l1 = ListNode(2)
l1.next = ListNode(4)
l1.next.next = ListNode(3)
l2 = ListNode(5)
l2.next = ListNode(6)
l2.next.next = ListNode(4)
result = addTwoNumbers(l1, l2)
while result:
    print(result.val, end=" ")
    result = result.next

```

Output:

```
7 0 8
```

3. 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 substrin

Program:

```

def lengthOfLongestSubstring(s):
    char_index = {} # Dictionary to store the index of each character
    max_length = 0
    start = 0
    for end, char in enumerate(s):
        if char in char_index and char_index[char] >= start:
            start = char_index[char] + 1
        char_index[char] = end
        max_length = max(max_length, end - start + 1)
    return max_length

```

```
s1 = "abcabcbb"
```

```

print(lengthOfLongestSubstring(s1))

s2 = "bbbbbb"

print(lengthOfLongestSubstring(s2))

s3 = "pwwkew"

print(lengthOfLongestSubstring(s3))

```

Output:

```

length of s1 is : 3
length of s2 is : 1
length of s2 is : 3

```

4. 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 $(2 + 3) / 2 = 2.5$

Program:

```

def findMedianSortedArrays(nums1, nums2):
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1
    m, n = len(nums1), len(nums2)
    low, high = 0, m
    while low <= high:
        partitionX = (low + high) // 2
        partitionY = (m + n + 1) // 2 - partitionX
        maxLeftX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
        minRightX = float('inf') if partitionX == m else nums1[partitionX]
        maxLeftY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
        minRightY = float('inf') if partitionY == n else nums2[partitionY]
        if maxLeftX <= minRightY and maxLeftY <= minRightX:
            if (m + n) % 2 == 0:
                return (max(maxLeftX, maxLeftY) + min(minRightX, minRightY)) / 2
            else:

```

```

        return max(maxLeftX, maxLeftY)
    elif maxLeftX > minRightY:
        high = partitionX - 1
    else:
        low = partitionX + 1
nums1 = [1, 3]
nums2 = [2]
print(findMedianSortedArrays(nums1, nums2))
nums3 = [1, 2]
nums4 = [3, 4]
print(findMedianSortedArrays(nums3, nums4))

```

Output:

```

2
2.5

```

5. 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"

Program:

```

def longestPalindrome(s):
    if len(s) < 2:
        return s

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

    longest = ""
    for i in range(len(s)):
        palindrome1 = expand_around_center(i, i)
        if len(palindrome1) > len(longest):
            longest = palindrome1

```

```

        palindrome2 = expand_around_center(i, i + 1)

    if len(palindrome2) > len(longest):

        longest = palindrome2

    return longest

s1 = "babad"

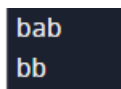
print(longestPalindrome(s1))

s2 = "cbbd"

print(longestPalindrome(s2))

```

Output:



```

bab
bb

```

6. 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"

Program:

```

def convert(s, numRows):

    if numRows == 1 or numRows >= len(s):

        return s

    rows = [""] * numRows

    index, step = 0, 1

    for char in s:

        rows[index] += char

        if index == 0:

            step = 1

        elif index == numRows - 1:

            step = -1

        index += step

    return "".join(rows)

```

```

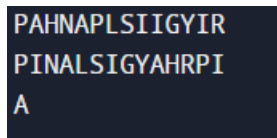
s1 = "PAYPALISHIRING"
numRows1 = 3
print(convert(s1, numRows1))

s2 = "PAYPALISHIRING"
numRows2 = 4
print(convert(s2, numRows2))

s3 = "A"
numRows3 = 1
print(convert(s3, numRows3))

```

Output:



```

PAHNAPLSIIGYIR
PINALSIGYAHRPI
A

```

7. 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

Program:

```

def reverse(x):
    sign = 1
    if x < 0:
        sign = -1
    x = abs(x)
    reversed_str = str(x)[::-1]
    reversed_int = int(reversed_str) * sign
    if reversed_int < -2**31 or reversed_int > 2**31 - 1:
        return 0
    return reversed_int

x1 = 123
print(reverse(x1))

```

```

x2 = -123

print(reverse(x2))

x3 = 120

print(reverse(x3))

```

Output:

```

321
-321
21

```

8. 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

Program:

```

def myAtoi(s):
    s = s.lstrip()
    sign = 1
    if s and (s[0] == '+' or s[0] == '-'):
        sign = -1 if s[0] == '-' else 1
    s = s[1:]

```



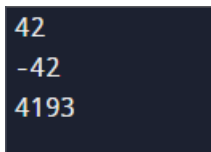
```

num = 0
for char in s:
    if char.isdigit():
        num = num * 10 + int(char)
    else:
        break
num *= sign
INT_MIN, INT_MAX = -2**31, 2**31 - 1
num = max(INT_MIN, min(INT_MAX, num))
return num

s1 = "42"
print(myAtoi(s1))
s2 = "-42"
print(myAtoi(s2))
s3 = "4193 with words"
print(myAtoi(s3))

```

Output:



```

42
-42
4193

```

9. 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.

Program:

```

def isPalindrome(x):
    if x < 0 or (x % 10 == 0 and x != 0):
        return False

```

```

reversed_num = 0
original_num = x
while x > 0:
    digit = x % 10
    reversed_num = reversed_num * 10 + digit
    x //= 10
return original_num == reversed_num

x1 = 121
print(isPalindrome(x1))

x2 = -121
print(isPalindrome(x2))

x3 = 10
print(isPalindrome(x3))

```

Output:

```

True
False
False

```

10. 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 (.)"

Program:

```

def isMatch(s, p):
    dp = [[False] * (len(p) + 1) for _ in range(len(s) + 1)]
    dp[0][0] = True
    for j in range(2, 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] == '.' or p[j - 1] == s[i - 1]:
            dp[i][j] = dp[i - 1][j - 1]
        elif p[j - 1] == '*':
            dp[i][j] = dp[i][j - 2] or (dp[i - 1][j] and (p[j - 2] == '.' or p[j - 2] == s[i - 1]))
    return dp[len(s)][len(p)]

s1, p1 = "aa", "a"
print(isMatch(s1, p1))

s2, p2 = "aa", "a*"
print(isMatch(s2, p2))

s3, p3 = "ab", ".*"
print(isMatch(s3, p3))

```

Output:

```

False
True
True

```