**DATA STRCTURES**

**LINKED LIST:** class Node: def \_\_init\_\_(self, data): # Corrected the init method self.data = data

self.next = None

class LinkedList: def \_\_init\_\_(self): # Corrected the init method self.head = None

def insertAtBegin(self, data): new\_node = Node(data) if self.head is None:

self.head = new\_node else:

new\_node.next = self.head

self.head = new\_node

def insertAtIndex(self, data, index):

new\_node = Node(data) current\_node = self.head position = 0 if position == index:

self.insertAtBegin(data) else: while current\_node is not None and position + 1 != index:

position += 1

current\_node = current\_node.next

if current\_node is not None: new\_node.next = current\_node.next current\_node.next = new\_node else:

print("Index not present")

def insertAtEnd(self, data): new\_node = Node(data) if self.head is None:

self.head = new\_node return

current\_node = self.head while current\_node.next: current\_node = current\_node.next

current\_node.next = new\_node

def remove\_first\_node(self): if self.head is None:

return

self.head = self.head.next

def remove\_last\_node(self): if self.head is None:

return

if self.head.next is None: # Special case if there is only one element

self.head = None

return

current\_node = self.head while current\_node.next.next:

current\_node = current\_node.next

current\_node.next = None

def remove\_at\_index(self, index):

if self.head is None:

return

current\_node = self.head position = 0 if position == index: self.remove\_first\_node() else: while current\_node is not None and position + 1 != index:

position += 1

current\_node = current\_node.next

if current\_node is not None and current\_node.next is not None:

current\_node.next = current\_node.next.next else:

print("Index not present")

def remove\_node(self, data): current\_node = self.head

if current\_node is not None and current\_node.data == data:

self.remove\_first\_node()

return

prev\_node = None while current\_node is not None and current\_node.data != data: prev\_node = current\_node

current\_node = current\_node.next

if current\_node is not None:

prev\_node.next = current\_node.next else:

print("Node with data not found")

def sizeofLL(self):

size = 0 current\_node = self.head while current\_node:

size += 1

current\_node = current\_node.next

return size

def printLL(self):

current\_node = self.head while current\_node:

print(current\_node.data)

current\_node = current\_node.next

llist = LinkedList()

llist.insertAtEnd('a') llist.insertAtEnd('b') llist.insertAtBegin('c') llist.insertAtEnd('d')

llist.insertAtIndex(data='g', index=2)

print("Node Data")

llist.printLL()

print("\nRemove First Node") llist.remove\_first\_node()

llist.printLL()

print("\nRemove Last Node") llist.remove\_last\_node()

llist.printLL()

**OUTPUT:**

Node Data

c a g b d

Remove First Node

a g b d

Remove Last Node

a g b

**DOUBLE LINKED LIST:** class Node: def \_init\_(self, data=None):

self.data = data self.next = None self.prev = None

class DoublyLinkedList:

def \_init\_(self): self.head = None self.tail = None

def is\_empty(self):

return self.head is None

def append(self, data): new\_node = Node(data) if self.is\_empty(): self.head = new\_node self.tail = new\_node else:

new\_node.prev = self.tail self.tail.next = new\_node self.tail = new\_node

def prepend(self, data): new\_node = Node(data) if self.is\_empty(): self.head = new\_node self.tail = new\_node else:

new\_node.next = self.head self.head.prev = new\_node self.head = new\_node

def delete(self, data): if self.is\_empty():

return

current = self.head while current is not None and current.data != data:

current = current.next

if current is not None: if current.prev is not None:

current.prev.next = current.next

else:

self.head = current.next

if current.next is not None:

current.next.prev = current.prev

else:

self.tail = current.prev

def display\_forward(self):

elements = [] current = self.head while current:

elements.append(current.data) current = current.next print(" -> ".join(map(str, elements)))

def display\_backward(self):

elements = [] current = self.tail while current:

elements.append(current.data) current = current.prev print(" -> ".join(map(str, elements)))

# Example usage:

my\_doubly\_linked\_list = DoublyLinkedList() my\_doubly\_linked\_list.append(1) my\_doubly\_linked\_list.append(2) my\_doubly\_linked\_list.append(3) my\_doubly\_linked\_list.prepend(0)

my\_doubly\_linked\_list.display\_forward() # Output: 0 -> 1 -> 2 -> 3 my\_doubly\_linked\_list.display\_backward() # Output: 3 -> 2 -> 1 -> 0

my\_doubly\_linked\_list.delete(2)

my\_doubly\_linked\_list.display\_forward() # Output: 0 -> 1 -> 3 my\_doubly\_linked\_list.display\_backward() # Output: 3 -> 1 -> 0

**OUTPUT:**

0 -> 1 -> 2 -> 3

3 -> 2 -> 1 -> 0

0 -> 1 -> 3

3 -> 1 -> 0

**CIRCULAR LINKED LIST:** class Node: def \_\_init\_\_(self, data=None): # Corrected the init method self.data = data self.next = None

class CircularLinkedList: def \_\_init\_\_(self): # Corrected the init method self.head = None

def is\_empty(self):

return self.head is None

def append(self, data): new\_node = Node(data) if self.is\_empty():

self.head = new\_node new\_node.next = self.head else:

current = self.head while current.next != self.head:

current = current.next current.next = new\_node new\_node.next = self.head

def prepend(self, data): new\_node = Node(data) if self.is\_empty(): self.head = new\_node new\_node.next = self.head

else:

current = self.head while current.next != self.head:

current = current.next current.next = new\_node new\_node.next = self.head self.head = new\_node

def delete(self, data): if self.is\_empty():

return

if self.head.data == data: current = self.head while current.next != self.head:

current = current.next if self.head.next == self.head:

self.head = None else:

self.head = self.head.next current.next = self.head else:

current = self.head while current.next != self.head and current.next.data != data:

current = current.next

if current.next.data == data:

current.next = current.next.next

def display(self): elements = [] current = self.head if current:

repeat = True while repeat:

elements.append(current.data) current = current.next if current == self.head:

repeat = False

print(" -> ".join(map(str, elements)))

def search(self, target): if self.is\_empty():

return False

current = self.head repeat = True while repeat: if current.data == target:

return True

current = current.next if current == self.head:

repeat = False

return False

# Example usage: my\_circular\_linked\_list = CircularLinkedList() my\_circular\_linked\_list.append(1) my\_circular\_linked\_list.append(2) my\_circular\_linked\_list.append(3) my\_circular\_linked\_list.prepend(0)

my\_circular\_linked\_list.display() # Output: 0 -> 1 -> 2 -> 3

my\_circular\_linked\_list.delete(2) my\_circular\_linked\_list.display() # Output: 0 -> 1 -> 3

**OUTPUT:**

0 -> 1 -> 2 -> 3

0 -> 1 -> 3

**LEET CODE 21:**

# Definition for singly-linked list.

# class ListNode:

#     def \_init\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:

        newHead = dummyHead = ListNode()

        while list1 and list2:

            if list1.val < list2.val:

                dummyHead.next = list1

                list1 = list1.next

            else:

                dummyHead.next = list2

                list2 = list2.next

            dummyHead = dummyHead.next

        if list1:

            dummyHead.next = list1

        if list2:

            dummyHead.next = list2

        return newHead.next

**OUTPUT:**

list1 =

[1,2,4]

list2 =

[1,3,4]

1

[1,2,4]

Source

**LEETCODE 1290:**

Given head which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the *decimal value* of the number in the linked list.

The **most significant bit** is at the head of the linked list.

**Example 1:**



**Input:** head = [1,0,1]

**Output:** 5

**Explanation:** (101) in base 2 = (5) in base 10

**Example 2:**

**Input:** head = [0]

**Output:** 0

**Constraints:**

* The Linked List is not empty.
* Number of nodes will not exceed 30.
* Each node's value is either 0 or 1.

# Definition for singly-linked list.

# class ListNode:

#     def \_init\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def getDecimalValue(self, head: ListNode) -> int:

        answer = ''

        if head is None:

            return

        current = head

        while current:

            answer = answer + str(current.val)

            current = current.next

        return int(answer,2)

**OUTPUT:**

Input

head =

[1,0,1]

Output

5

Expected

5

**LEETCODE 876:**

Given the head of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

**Example 1:**

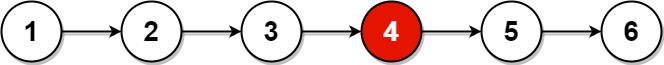


**Input:** head = [1,2,3,4,5]

**Output:** [3,4,5]

**Explanation:** The middle node of the list is node 3.

**Example 2:**



**Input:** head = [1,2,3,4,5,6]

**Output:** [4,5,6]

**Explanation:** Since the list has two middle nodes with values 3 and 4, we return the second one.

**Constraints:**

* The number of nodes in the list is in the range [1, 100].
* 1 <= Node.val <= 100

# Definition for singly-linked list.

# class ListNode:

#     def \_init\_(self, val=0, next=None):

#         self.val = val

#         self.next = next

class Solution:

    def middleNode(self, head):

        slow\_pointer = head

        fast\_pointer = head

        while fast\_pointer is not None and fast\_pointer.next is not None:

            slow\_pointer = slow\_pointer.next

            fast\_pointer = fast\_pointer.next.next

        return slow\_pointer

**OUTPUT:**

Input

head =

[1,2,3,4,5]

Output

[3,4,5]

Expected

[3,4,5]

**LEETCODE 771:**

You're given strings jewels representing the types of stones that are jewels, and stones representing the stones you have. Each character in stones is a type of stone you have. You want to know how many of the stones you have are also jewels.

Letters are case sensitive, so "a" is considered a different type of stone from "A".

**Example 1:**

**Input:** jewels = "aA", stones = "aAAbbbb"

**Output:** 3

**Example 2:**

**Input:** jewels = "z", stones = "ZZ"

**Output:** 0

**Constraints:**

* 1 <= jewels.length, stones.length <= 50
* jewels and stones consist of only English letters.
* All the characters of jewels are **unique**.

    def numJewelsInStones(self, jewels: str, stones: str) -> int:

        jew\_list = list(jewels)

        answer = 0

        for j in jew\_list:

            answer += stones.count(j)

        return answer

**OUTPUT:**

Input

jewels =

"aA"

stones =

"aAAbbbb"

Output

3

Expected

3

**LEET CODE 2744:**

You are given a **0-indexed** array words consisting of **distinct** strings.

The string words[i] can be paired with the string words[j] if:

* The string words[i] is equal to the reversed string of words[j].
* 0 <= i < j < words.length.

Return *the****maximum****number of pairs that can be formed from the array*words*.*

Note that each string can belong in **at most one** pair.

**Example 1:**

**Input:** words = ["cd","ac","dc","ca","zz"]

**Output:** 2

**Explanation:** In this example, we can form 2 pair of strings in the following way:

- We pair the 0th string with the 2nd string, as the reversed string of word[0] is "dc" and is equal to words[2].

- We pair the 1st string with the 3rd string, as the reversed string of word[1] is "ca" and is equal to words[3].

It can be proven that 2 is the maximum number of pairs that can be formed.

**Example 2:**

**Input:** words = ["ab","ba","cc"]

**Output:** 1

**Explanation:** In this example, we can form 1 pair of strings in the following way:

- We pair the 0th string with the 1st string, as the reversed string of words[1] is "ab" and is equal to words[0].

It can be proven that 1 is the maximum number of pairs that can be formed.

**Example 3:**

**Input:** words = ["aa","ab"]

**Output:** 0

**Explanation:** In this example, we are unable to form any pair of strings.

class Solution:

    def maximumNumberOfStringPairs(self, words: List[str]) -> int:

        result = 0

        for i in range(len(words)):

            if words[i][::-1] in (words[i+1:]):

                result+=1

        return result

**OUTPUT:**

Input

words =

["cd","ac","dc","ca","zz"]

Output

2

Expected

2

**LEETCODE 2418:**

You are given an array of strings names, and an array heights that consists of **distinct** positive integers. Both arrays are of length n.

For each index i, names[i] and heights[i] denote the name and height of the ith person.

Return names*sorted in****descending****order by the people's heights*.

**Example 1:**

**Input:** names = ["Mary","John","Emma"], heights = [180,165,170]

**Output:** ["Mary","Emma","John"]

**Explanation:** Mary is the tallest, followed by Emma and John.

**Example 2:**

**Input:** names = ["Alice","Bob","Bob"], heights = [155,185,150]

**Output:** ["Bob","Alice","Bob"]

**Explanation:** The first Bob is the tallest, followed by Alice and the second Bob.

**Constraints:**

* n == names.length == heights.length
* 1 <= n <= 103
* 1 <= names[i].length <= 20
* 1 <= heights[i] <= 105
* names[i] consists of lower and upper case English letters.
* All the values of heights are distinct.

class Solution:

    def sortPeople(self, names: List[str], heights: List[int]) -> List[str]:

        ans=zip(heights,names)

        l=[]

        for i,j in sorted(ans):

            l.append(j)

        return l[::-1]

**OUTPUT:**

Input

names =

["Mary","John","Emma"]

heights =

[180,165,170]

Output

["Mary","Emma","John"]

Expected

["Mary","Emma","John"]

**LEETCODE 1342:**

Given an integer num, return *the number of steps to reduce it to zero*.

In one step, if the current number is even, you have to divide it by 2, otherwise, you have to subtract 1 from it.

**Example 1:**

**Input:** num = 14

**Output:** 6

**Explanation:**

Step 1) 14 is even; divide by 2 and obtain 7.

Step 2) 7 is odd; subtract 1 and obtain 6.

Step 3) 6 is even; divide by 2 and obtain 3.

Step 4) 3 is odd; subtract 1 and obtain 2.

Step 5) 2 is even; divide by 2 and obtain 1.

Step 6) 1 is odd; subtract 1 and obtain 0.

**Example 2:**

**Input:** num = 8

**Output:** 4

**Explanation:**

Step 1) 8 is even; divide by 2 and obtain 4.

Step 2) 4 is even; divide by 2 and obtain 2.

Step 3) 2 is even; divide by 2 and obtain 1.

Step 4) 1 is odd; subtract 1 and obtain 0.

**Example 3:**

**Input:** num = 123

**Output:** 12

**Constraints:**

* 0 <= num <= 106

class Solution:

    def numberOfSteps(self, n: int) -> int:

        step=0

        while n!=0:

            if n%2==0:

                n/=2

                step+=1

            else:

                n-=1

                step+=1

        return step

**OUTPUT:**

Input

num =

14

Output

6

Expected

6

**LEET CODE 3099**

An integer divisible by the **sum** of its digits is said to be a **Harshad** number. You are given an integer x. Return*the sum of the digits*ofxifxis a **Harshad** number, otherwise, return-1*.*

**Example 1:**

**Input:** x = 18

**Output:** 9

**Explanation:**

The sum of digits of x is 9. 18 is divisible by 9. So 18 is a Harshad number and the answer is 9.

**Example 2:**

**Input:** x = 23

**Output:** -1

**Explanation:**

The sum of digits of x is 5. 23 is not divisible by 5. So 23 is not a Harshad number and the answer is -1.

**Constraints:**

* 1 <= x <= 100

class Solution:

    def sumOfTheDigitsOfHarshadNumber(self, x: int) -> int:

        n=x

        result=0

        while n!=0:

            result +=(n%10)

            n=n//10

        if x%result == 0:

            return result

        else:

            return - 1

**OUTPUT:**

Input

x =

18

Output

9

Expected

9