------------------------------------------------------------------------

Ejercicios listas enlazadas

------------------------------------------------------------------------

La explicación de este ejercicio es parecida al que le sigue, ya que se recorre la lista de inicio a final hasta que se encuentre la posición a eliminar, pero se cambian algunas cosas, si la posición a eliminar es 0, entonces se elimina la cabeza, esto indica que la cabeza nueva será la posición siguiente a la cabeza. Además de que cambia la lógica de apuntadores después de encontrar la posición, ya que una vez encontrada basta con que el nodo anterior apunte a el nodo siguiente del que se quiere eliminar, es decir al siguiente siguiente del anterior a eliminar

**DELETE A NODE**

def deleteNode(head, position):

    if position == 0:

        head = head.next

    else:

        aux = head

        contador = 1

        while aux is not None and contador < position:

            temp = temp.next

            contador += 1

        aux.next = aux.next.next

    return head

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

La explicación de este ejercicio es primero asignar un caso base donde si la lista es vacía el valor que nos dan será el primer valor de la lista. En los demás casos lo que se hace es recorrer la lista hasta que se llegue al final o que el contador asignado sea igual a la posición dada, una vez se haya encontrado la posición, se acomodan los apuntadores de forma que el anterior apunte a el nodo a ingresar y el nodo a ingresar apunte al nodo siguiente.

**INSERT A NODE AT A SPECIFIC POSITION**

def insertNodeAtPosition(llist, data, position):

    node = SinglyLinkedListNode(data)

    if llist == None:

        llist = node

    else:

        aux = llist

        contador = 1

        while aux is not None and contador < position:

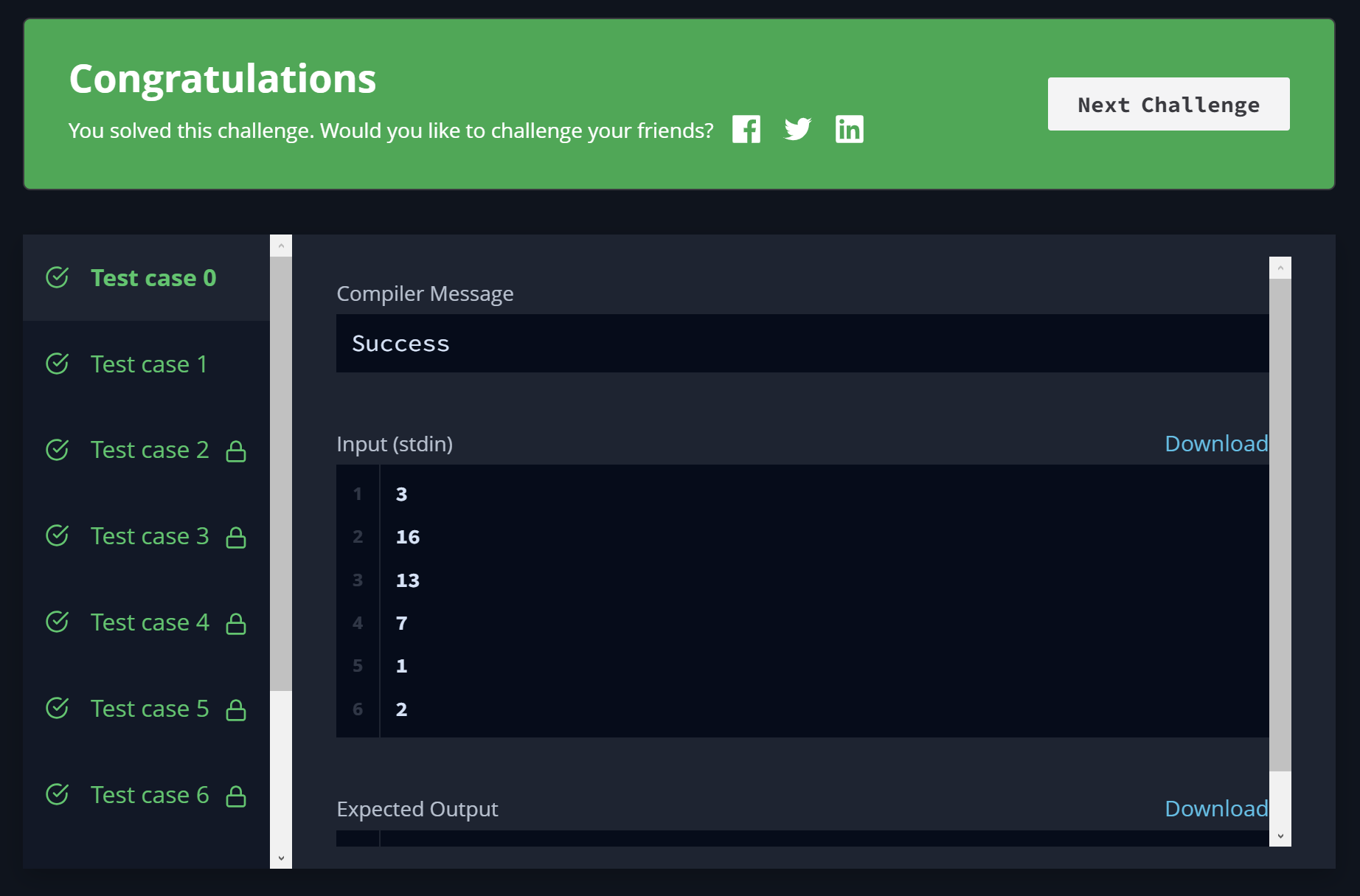
            aux = aux.next

            contador += 1

        node.next = aux.next

        aux.next = node

    return llist



Par este ejercicio primero se haya cual es la lista más larga para así iterar sobre ella, pero no se descarta la longitud de la lista2, ya que se necesita hallar la diferencia porque ahí se intersecan las listas. Una vez se tenga la intersección, se recorre la lista más larga hasta el punto de intersección y allí se comprueba que los nodos efectivamente sean iguales, en caso de que no, se avanza en ambas cabezas de las 2 listas en

**FIND MERGE POINT OF TWO MERGED LISTS**

def findMergeNode(head1, head2):

    def longitudes(head):

        length = 0

        while head.next is not None:

            head = head.next

            length += 1

        return length

    def comun(diff,head1,head2):

        for i in range(diff):

            head1 = head1.next

        while head1 and head2:

            if head1 == head2:

                return head1.data

            else:

                head1 = head1.next

                head2 = head2.next

    len1 = longitudes(head1)

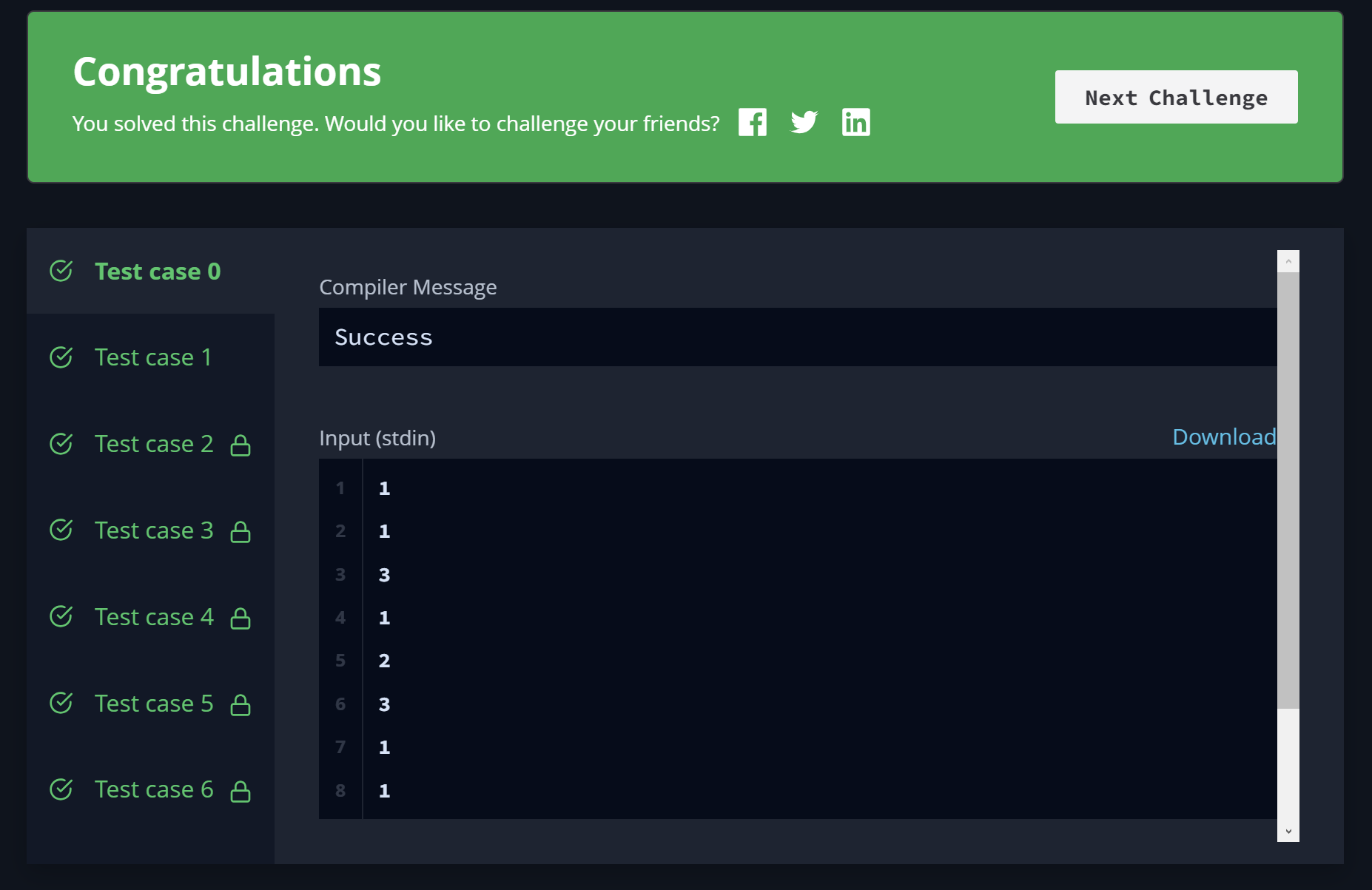
    len2 = longitudes(head2)

    if len1 > len2:

        return comun(len1 - len2,head1,head2)

    else:

        return comun(len2 - len1,head2,head1)



Para la explicación del código se toma que la lista para este ejercicio es doblemente enlazada y por ende tiene apuntadores prev y next, lo que se hace es cambiar los apuntadores de dirección así se “voltea la lista”, y el apuntador inicial de la cabeza apuntará al que en ese momento es la “cola”, al final del código se cambian los apuntadores de la nueva cabeza.

**REVERSE A DOUBLY LINKED LIST**

def reverse(llist):

    while llist.next is not None:

        llist.next,llist.prev,llist = llist.prev,llist.next,llist.next

    llist.next,llist.prev = llist.prev,None

    return llist

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

Para la explicación de este código, se asigna una variable que apunta a la cabeza menor entre las 2 listas y retorna ese valor, hasta que una de las cabezas es vacía y retorna la cabeza de la lista resultante

**MERGE TWO SORTED LINKED LISTS**

def mergeLists(head1, head2):

    if head1 == head2 == None:

        return None

    if head1 == None:

        return head2

    if head2 == None:

        return head1

    if head1.data < head2.data:

        aux = head1

        aux.next = mergeLists(head1.next,head2)

    else:

        aux = head2

        aux.next = mergeLists(head1,head2.next)

    return aux

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

------------------------------------------------------------------------

Ejercicios árboles

------------------------------------------------------------------------

La explicación de este código es la siguiente, primero como el valor que entra es un valor, no un nodo, se debe crear un nodo, el cual su valor sea el que se ingresa, ahora se hace el proceso de inserción ya visto; se revisa si el árbol está vacío y si es así el primer valor que entra será la raíz. Ahora se ve si el valor a ingresar es mayor o menor que la raíz, así sabemos donde se ingresará, el proceso se repetirá hasta que actual sea None, es decir que allí se podrá insertar el valor y retorna el valor para imprimirlo.

**INSERTION**

class Node:

    def \_\_init\_\_(self,value):

        self.value = value

        self.left = None

        self.right = None

    def insert(self, val):

        new\_node = Node(val)

        if self.root is None:

            self.root = new\_node

            return

        actual = self.root

        while actual:

            if val < actual.info:

                if actual.left is None:

                    actual.left = new\_node

                    return

                actual = actual.left

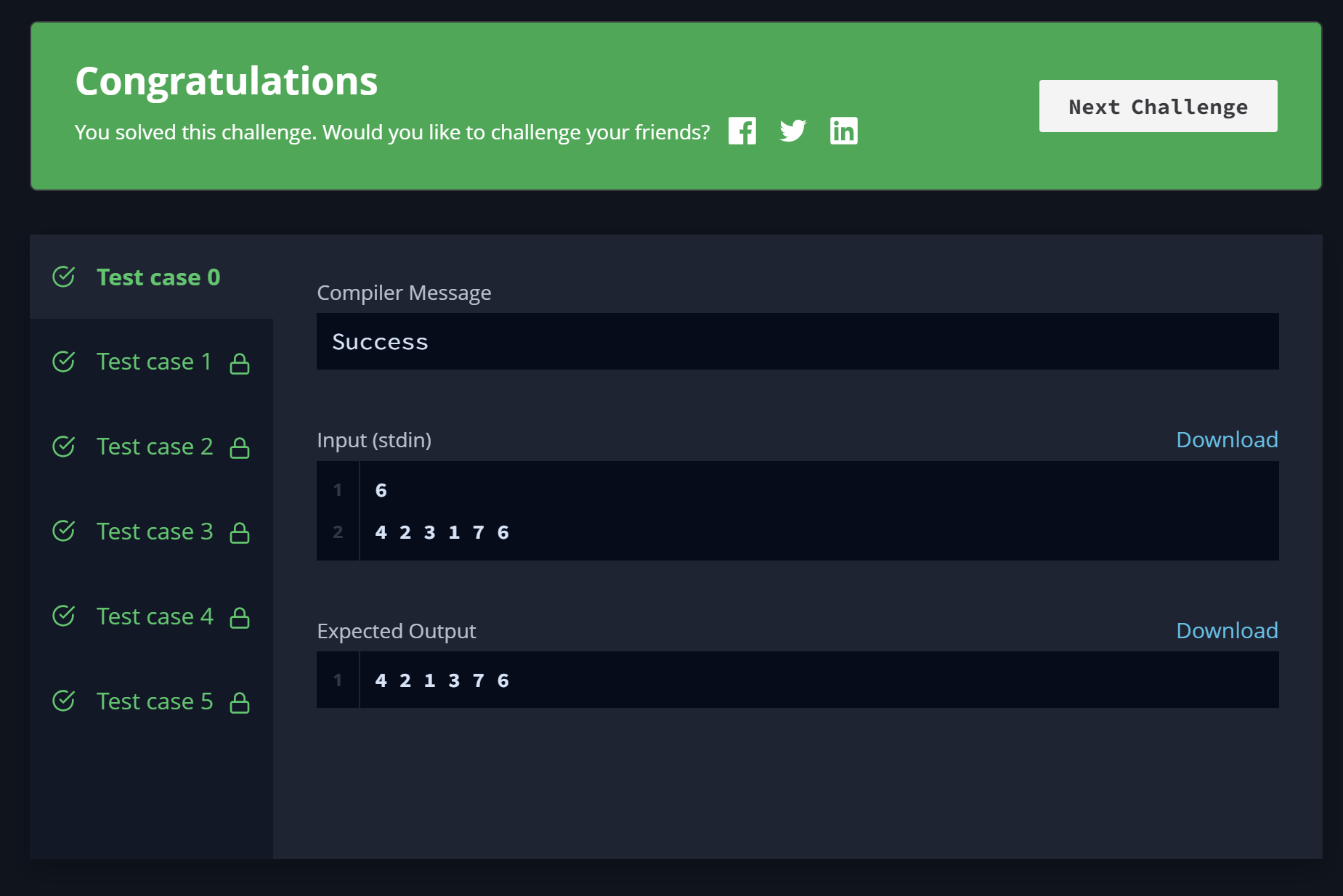
            else:

                if actual.right is None:

                    actual.right = new\_node

                    return

                actual = actual.right



Este código es bastante fácil de explicar ya que es el recorrido en postOrder de los árboles de búsqueda binarios, el cual por definición es recorrer el nodo izquierdo, luego el derecho y por último la raíz.

**POST ORDER TRAVERSAL**

class Node:

    def \_\_init\_\_(self,value):

        self.value = value

        self.left = None

        self.right = None

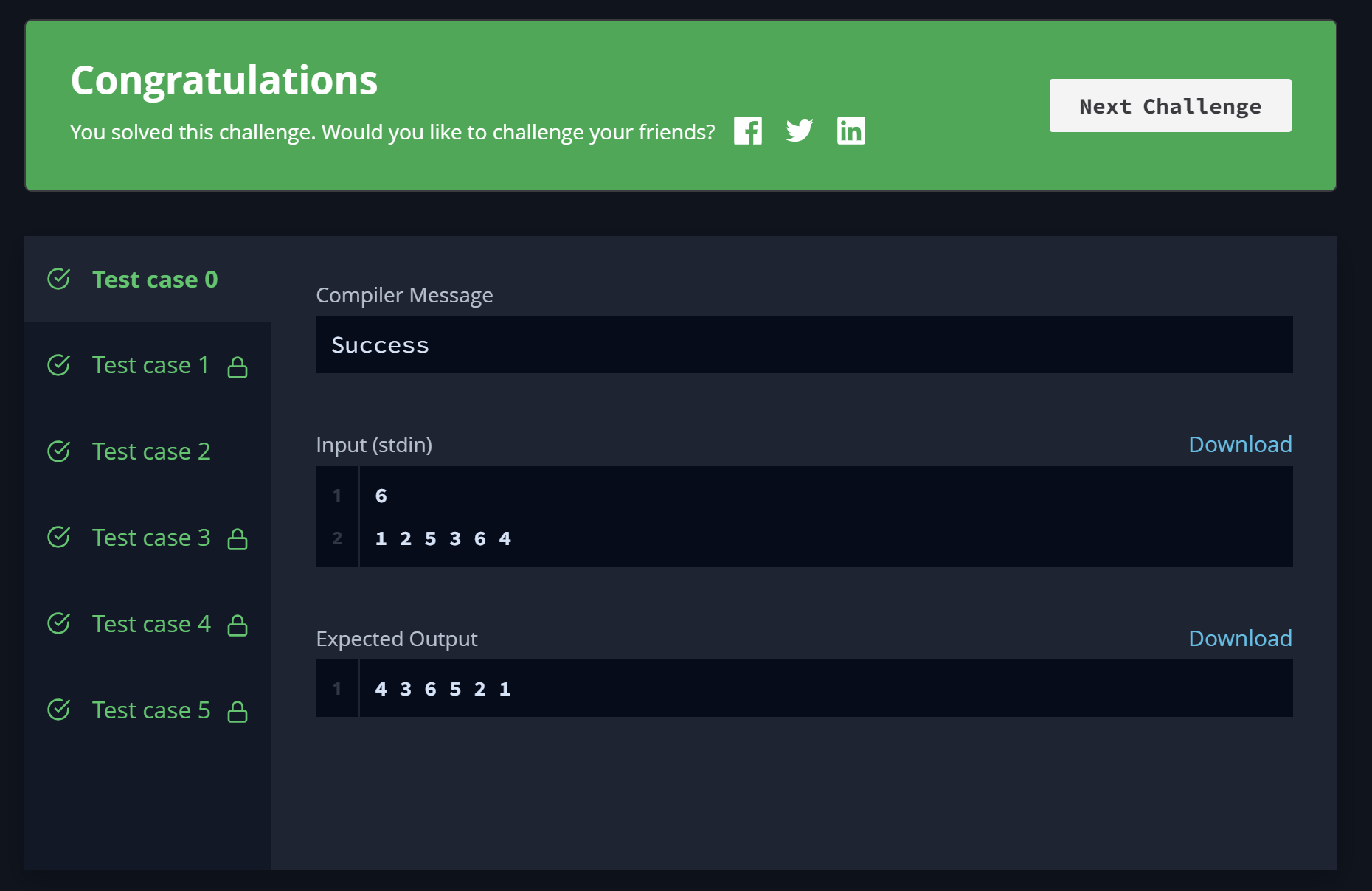
def postOrder(root):

    if root:

        postOrder(root.left)

        postOrder(root.right)

        print(root, end = " ")



La explicación del problema se basa en un diccionario que guarda por cada nodo el nivel en el que se encuentra como si fuera una matriz, donde i es la distancia a la raíz y j es el level, por lo tanto primero se mira que el nodo no esté en el diccionario y se agrega, de lo contrario si se encuentra ya, se revisa si el nivel que tiene es mayor que el que había guardado, de ser así se actualiza la llave del diccionario y mediante recurrencia se va hacia la izquierda o derecha del árbol y se resta la distancia a la raíz o se suma según si se va hacia derecha o izquierda.

**TREE TOP VIEW**

class Node:

    def \_\_init\_\_(self,value):

        self.value = value

        self.left = None

        self.right = None

def topView(root):

    d = {}

    def viaje(root,key,level):

        if root:

            if key not in d:

                d[key] = [root,level]

            elif d[key][1] > level:

                d[key] = root,level

            viaje(root.left,key - 1,level + 1)

            viaje(root.right,key +1,level + 1)

    viaje(root,0,0)

    for \_ in sorted(d):

        print(d[\_][0],end= " ")

Captura de pantalla de un celular

Descripción generada automáticamente

La explicación del código es bastante simple, por definición de el recorrido inorder de un árbol, primero se recorre la raíz, después el nodo izquierdo y después el nodo derecho así sucesivamente hasta que se haya recorrido todo el árbol.

**TREE INORDER TRAVERSAL**

class Node:

    def \_\_init\_\_(self,value):

        self.value = value

        self.left = None

        self.right = None

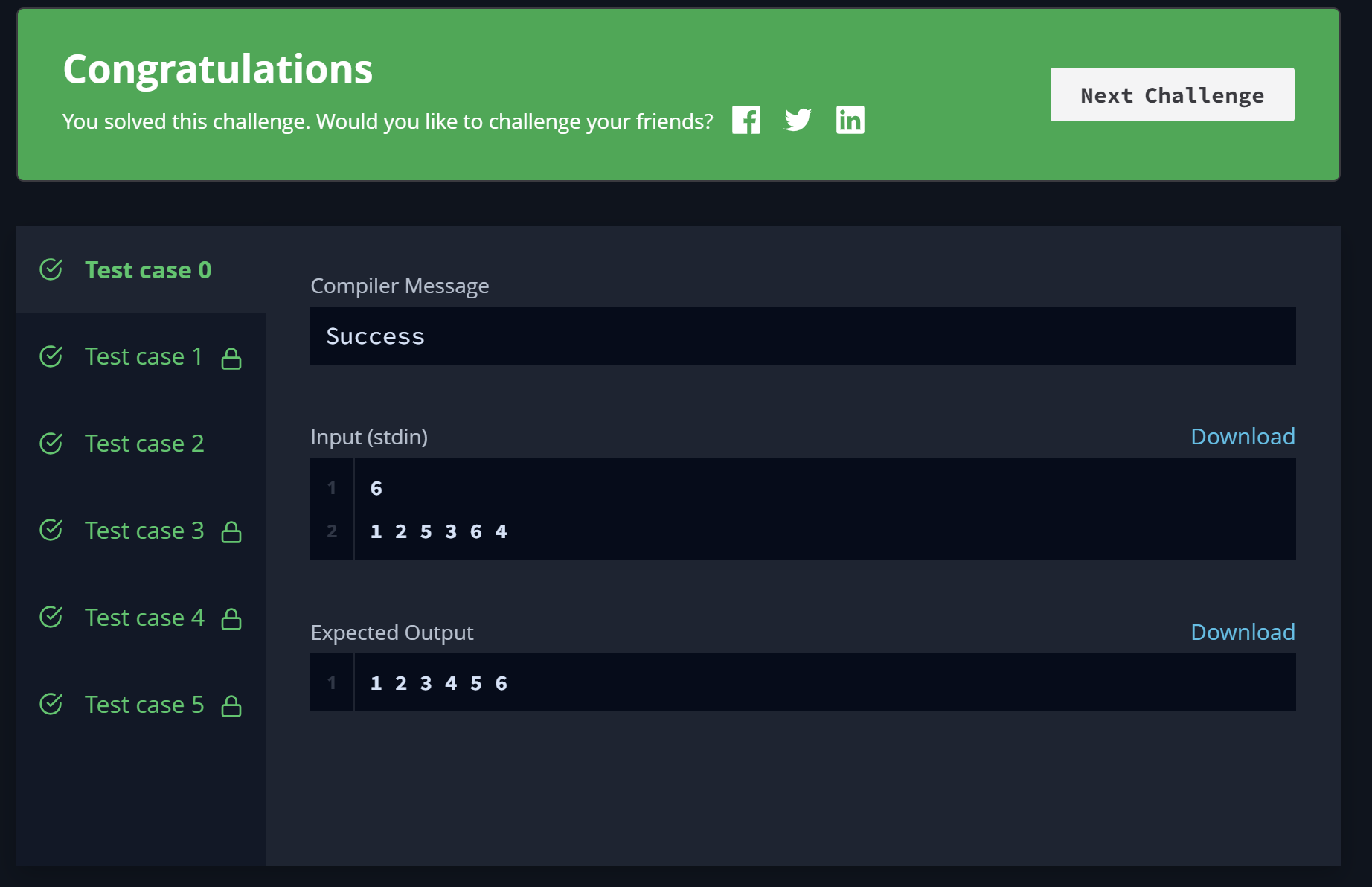
def inOrder(root):

    if root:

        inOrder(root.left)

        print(root, end = " ")

        inOrder(root.right)



La explicación del código es bastante simple, la definición de la clase nodo ya estaba hecha, por lo tanto simplemente se recorre el árbol desde el nodo de la raíz a la izquierda y derecha y se va sumando uno cada vez que se recorre, hasta que ya sea una hoja, la función retorna el máximo y así se halla la altura.

**TREE HEIGHT OF A BINARY TREE**

class Node:

    def \_\_init\_\_(self,value):

        self.value = value

        self.left = None

        self.right = None

def height(root):

    if root is None:

        return -1

    else:

        return 1 + max(height(root.left),height(root.right))

Captura de pantalla de un celular

Descripción generada automáticamente

------------------------------------------------------------------------

Ejercicios grafos

------------------------------------------------------------------------

La explicación del ejercicio EVEN TREE se basa en la lista inicializada en 0, que tiene de cantidad de valores, la cantidad de nodos, por otro lado, se crea la lista de adyacencia del grafo. Para el proceso principal se recorre el árbol mediante DFS para ver las relaciones que tiene cada nodo y contarlos, si la cantidad de nodos es par, se elimina, ya que al sumarlos con el nodo raíz, daría un número impar y esas son las ramas a eliminar, entonces se suma 1 a la cantidad de vértices eliminados.

**EVEN TREE**

def evenForest(t\_nodes, t\_edges, t\_from, t\_to):

    graph = [[] for i in range(t\_nodes + 1)]

    for x,y in zip(t\_from,t\_to):

        graph[x].append(y)

        graph[y].append(x)

    count = [0] \* (t\_nodes + 1)

    def dfs(u,graph,count):

        count[u] = 1

        removed\_edges = 0

        for v in graph[u]:

            if count[v] == 0:

                removed\_edges += dfs(v,graph,count)

                count[u] += count[v]

                if count[v] % 2 == 0:

                    removed\_edges += 1

        return removed\_edges

    return dfs(1,graph,count)

def main():

    t\_nodes, t\_edges = map(int, input().rstrip().split())

    t\_from = [0] \* t\_edges

    t\_to = [0] \* t\_edges

    for i in range(t\_edges):

        t\_from[i], t\_to[i] = map(int, input().rstrip().split())

    res = evenForest(t\_nodes, t\_edges, t\_from, t\_to)

    print(res)

main()

Interfaz de usuario gráfica, Aplicación, Sitio web

Descripción generada automáticamente

La explicación del ejercicio CUT THE TREE se basa primero en una función que genera el grafo correspondiente, mientras que en variables globales se guardan los siguientes valores; el peso total se va actualizando con el peso de cada nodo, el cual es el peso de si mismo más el de los hijos, los nodos padres para hacer la suma correcta del peso total y asegurarnos que si se suman los pesos de los hijos y del padre correspondiente. Por último se encuentra la variable diferencia\_min, que se encarga de ir guardando la menor diferencia, se inicializa en el valor más grande posible el cual es infinito, para que la primera vez que encuentre una diferencia siempre sea menor.

Por otro lado para recorrer el grafo se usa DFS, para recorrer cada vértice y sus relaciones y hallar la diferencia con la fórmula del peso total – 2 veces el peso del vértice en el que se está

**CUT THE TREE**

peso\_total=dict()

nodo\_padre=dict()

diferencia\_min=[float('inf')]

def dfs\_iterative(G, s, weights):

    total = sum(weights)

    stack = [s]

    visited = set()

    while stack:

        vertice = stack[-1]

        if vertice in visited:

            peso\_total[vertice]=weights[vertice-1]

            for c in G[vertice]:

                if nodo\_padre.get(c,-1)==vertice:

                    peso\_total[vertice]+=peso\_total[c]

            diferencia\_min[0]=min(diferencia\_min[0],

            abs(total-2\*peso\_total[vertice]))

            stack.pop()

            continue

        visited.add(vertice)

        for vecino in G[vertice]:

            if vecino not in visited:

                nodo\_padre[vecino]=vertice

                stack.append(vecino)

    return visited

def setGraph(cant\_nodos, aristas):

    g=dict()

    for \_t,\_e in aristas:

        if \_t not in g.keys():

            g[\_t]=[\_e]

        else:

            g[\_t].append(\_e)

        if \_e not in g.keys():

            g[\_e]=[\_t]

        else:

            g[\_e].append(\_t)

    g.update({k:[] for k in range(1,cant\_nodos+1) if k not in g.keys()})

    return g

def cutTheTree(nodes, aristas):

    G=setGraph(len(nodes),aristas)

    print(G)

    dfs\_iterative(G,1,nodes)

    return diferencia\_min[0]

if \_\_name\_\_ == '\_\_main\_\_':

    n = int(input().strip())

    data = list(map(int, input().rstrip().split()))

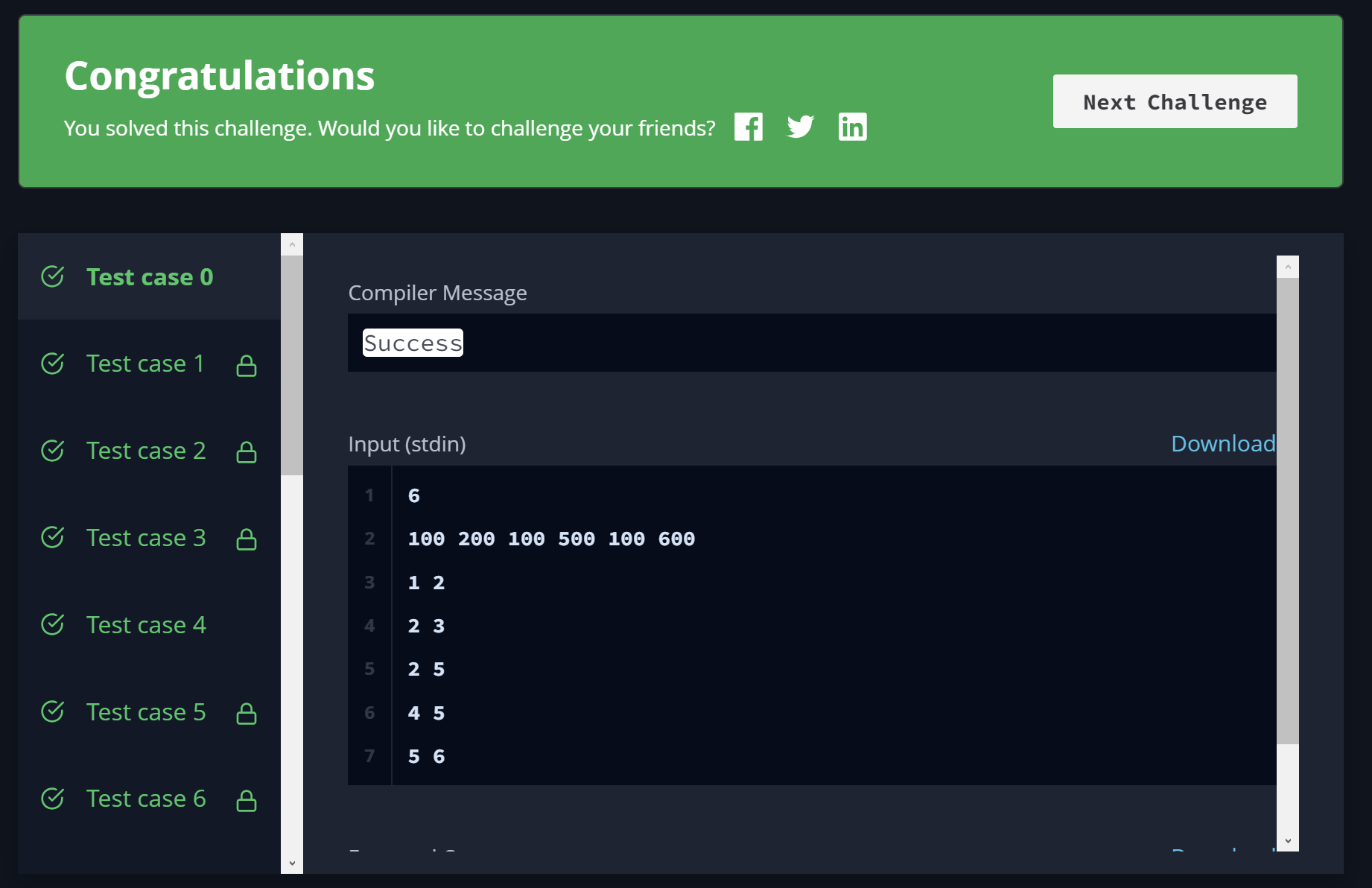
    aristas = []

    for \_ in range(n - 1):

        aristas.append(list(map(int, input().rstrip().split())))

    result = cutTheTree(data, aristas)

    print(str(result) + '\n')



La explicación del ejercicio BREADTH FIRST SEARCH SHORTEST REACH es primero crear el grafo adicionando los vértices a una lista de listas haciendo una lista de adyacencia, después inicializar 2 listas, una de las vértices visitados donde todos en un principio será False y el otro será la suma del peso de las relaciones, ya que entre cada nodo la relación es de 6, siempre se sumará 6 al peso, la lista de los pesos se inicializa con todos los vértices en -1 ya que es el valor a arrojar si un vértice es inalcanzable, por último se recorre la lista de adyacencia viendo las relaciones y sumando a las distancias 6 por cada relación.

**BFS SHORTEST REACH**

from collections import deque

def bfs(n,m,edges,s):

    graph = [[] for i in range(n + 1)]

    for x,y in edges:

        graph[x].append(y)

        graph[y].append(x)

    visited = [False] \* (n + 1)

    distances = [-1] \* (n + 1)

    q = deque([(s,0)])

    distances[s] = 0

    visited[s] = True

    while q:

        u,w = q.popleft()

        for v in graph[u]:

            if visited[v] == False:

                distances[v] = w + 6

                visited[v] = True

                q.append((v,w + 6))

    distances.remove(0)

    return distances[1:]

def main():

    q = int(input().strip())

    for q\_itr in range(q):

        first\_multiple\_input = input().rstrip().split()

        n = int(first\_multiple\_input[0])

        m = int(first\_multiple\_input[1])

        edges = []

        for \_ in range(m):

            edges.append(list(map(int, input().rstrip().split())))

        s = int(input().strip())

        result = bfs(n, m, edges, s)

        print(' '.join(map(str, result)))

main()

Interfaz de usuario gráfica

Descripción generada automáticamente

La explicación del ejercicio ROADS AND LIBRARIES es parecida a todos los demás ejercicios, ya que primero se genera la lista de adyacencia del grafo no dirigido, después se inicializa una variable que llevará la suma del costo total de construir las librerías y los caminos, como se usa la búsqueda DFS se inicia una lista con todos los valores en FALSE, en el algoritmo de DFS se inicializa el nodo inicial como TRUE y se empieza la búsqueda. En la función principal del ejercicio se manejan 2 costos, el costo por construirle una librería a la ciudad o de construir caminos que conecten a las demás ciudades, al final se retornará el mínimo y se encontrarán los costos mediante recursión.

**ROADS AND LIBRARIES**

def roadsAndLibraries(n, c\_lib, c\_road, cities):

    graph = [[] for i in range(n + 1)]

    for x,y in cities:

        graph[x].append(y)

        graph[y].append(x)

    costo = 0

    visitados = [False] \* (n + 1)

    def dfs(u,graph,visitados):

        visitados[u] = True

        ciudades = 1

        for v in graph[u]:

            if visitados[v] == False:

                ciudades += dfs(v,graph,visitados)

        return ciudades

    for v in range(1,n+1):

        if visitados[v] == False:

            ciudades = dfs(v,graph,visitados)

            costo1 = (ciudades - 1) \* c\_road + c\_lib

            costo2 = (ciudades) \* c\_lib

            costo += min(costo1,costo2)

    return costo

def main():

    q = int(input().strip())

    for q\_itr in range(q):

        first\_multiple\_input = input().rstrip().split()

        n = int(first\_multiple\_input[0])

        m = int(first\_multiple\_input[1])

        c\_lib = int(first\_multiple\_input[2])

        c\_road = int(first\_multiple\_input[3])

        cities = []

        for \_ in range(m):

            cities.append(list(map(int, input().rstrip().split())))

        result = roadsAndLibraries(n, c\_lib, c\_road, cities)

        print(str(result) + '\n')

Captura de pantalla de un celular

Descripción generada automáticamente

La estrategia para este ejercicio fue primero como todos los ejercicios, crear la lista de adyacencia para el grafo, primero se hayan todas las combinaciones posibles entre los astronautas, después se hayan los astronautas que son del mismo país e iteradamente se la van restando con la misma formula de n \* n – 1 // 2 hasta que ya se haya recorrido toda la lista de adyacencia. Este ejercicio no corrió uno de los casos ya que la recurrencia se va más del límite asignado por hackerrank.

**JOURNEY TO THE MOON**

def journeyToMoon(n, astronaut):

    graph = [[] for i in range(n)]

    for x,y in astronaut:

        graph[x].append(y)

        graph[y].append(x)

    visitados = [False] \* n

    parejas = n \* (n - 1) // 2

    def dfs(u,graph,visitados):

        visitados[u] = True

        vertices = 1

        for v in graph[u]:

            if visitados[v] == False:

                vertices += dfs(v,graph,visitados)

        return vertices

    for v in range(n):

        if visitados[v] == False:

            personas = dfs(v,graph,visitados)

            parejas -= personas \* (personas - 1) // 2

    return parejas

def main():

    first\_multiple\_input = input().rstrip().split()

    n = int(first\_multiple\_input[0])

    p = int(first\_multiple\_input[1])

    astronaut = []

    for \_ in range(p):

        astronaut.append(list(map(int, input().rstrip().split())))

    result = journeyToMoon(n, astronaut)

    print(str(result) + '\n')

