Estructura de Datos :3

Examen Ordinario

María Grissel Pérez Cortés

TEMA 3: CADENAS DE TEXTO

```python
# Ingrese un número entero - - - - - - - - - - - - - - - - - - - - - - - - - - -
numero_entero = int(input())
print (numero_entero)

# Ingrese un booleano (como "True" o "False") - - - - - - - - - - - - - - - - - -
booleano_str = input()

# - - - Convertir la entrada a un booleano
booleano = booleano_str.lower() == "true"
print(booleano)

# Ingrese su nombre y edad - - - - - - - - - - - - - - - - - - - - - - - - - - -
nombre = input()
edad = int(input())
print(f"{nombre}, {edad}")

# Recorrido básico de la cadena - - - - - - - - - - - - - - - - - - - - - - - - -
cadena = "Python"
for caracter in cadena:
    print(caracter)

# Recorrido inverso de la cadena
cadena = input()
for i in range(len(cadena)-1, -1, -1):
    print(cadena[i])

# Recorrido de la cadena con salto de dos caracteres - - - - - - - - - - - - - -
cadena = "ABCDEFGHIJ"
for i in range(0, len(cadena), 2):
    print(cadena[i])

# LEN: Obtener la longitud de una cadena de texto - - - - - - - - - - - - - - - -
cadena = "Hola mundo"
longitud = len(cadena)
print("La longitud de la cadena es:", longitud)  # Imprime: La longitud de la cadena
es: 10

# LEN: Obtener la longitud de una lista - - - - - - - - - - - - - - - - - - - - -
lista = [1, 2, 3, 4, 5, 6]
longitud = len(lista)
print("La longitud de la lista es:", longitud)  # Imprime: La longitud de la lista es:
6

# STRIP: Eliminar caracteres al principio y final - - - - - - - - - - - - - - - -
cadena = "---Hola mundo---"
limpia = cadena.strip("-")
print(limpia)  # Imprime: "Hola mundo"

# Dividir una cadena en palabras - - - - - - - - - - - - - - - - - - - - - - - -
cadena = "Hola mundo feliz"
palabras = cadena.split()
print(palabras)  # Imprime: ['Hola', 'mundo', 'feliz']
```

```python
# Caracteres individuales - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
cadena = "Python"
caracteres = list(cadena)
print(caracteres)  # Imprime: ['P', 'y', 't', 'h', 'o', 'n']
```

Ejercicios

```python
# Ejercicio: Palindromo - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def reversa (cadena):
    if not cadena:
        return ''
    res = ''
    for i in range (1, len(cadena)):
        res += cadena[-i]
    return res + cadena [0]
def es_palindromo(cadena):
    return cadena == reversa(cadena)
cadena = input()
print(es_palindromo(cadena))


# Ejercicio: Contar puntos - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def contar_puntos(cadena):
    cadena_limpia = cadena.strip()
    num_puntos = 0
    punto_anterior = False

    for caracter in cadena_limpia:
        if caracter == '.' and not punto_anterior:
            num_puntos += 1
            punto_anterior = True
        elif caracter != '.':
            punto_anterior = False
    return num_puntos
if __name__ == '__main__':
    entrada = input()
    resultado = contar_puntos(entrada)
    print(resultado)


# Extraer matrículas - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def obtener_matriculas(Num_cadenas, cadenas):
    return [cadena.split('(')[1].split(')')[0] for cadena in cadenas]
if __name__ == '__main__':
    Num_cadenas = int(input())
    cadenas = [input() for _ in range(Num_cadenas)]
    matriculas = obtener_matriculas(Num_cadenas, cadenas)
    for matricula in matriculas:
        print(matricula)

# Dar formato XML - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
entrada = input()
datos = entrada.split('.')
salida = f"<nombre>{datos[0]}</nombre><edad>{datos[1]}</edad><grado>{datos[2]}</grado>"
print(salida)


# Contar cuántas veces un prefijo específico - - - - - - - - - - - - - - - - - - -
n = int(input())
letra = input()
lista = [input() for _ in range(n)]
contador = sum(1 for palabra in lista if palabra.startswith(letra))
```

```
print(f"Cantidad de palabras que comienzan con {letra}:", contador)
```

TEMA 4: OBJETOS

```
# Clase Persona - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Persona():
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentarse(self):
        return f"¡Hola! Mi nombre es {self.nombre} y tengo {self.edad} años."

# Crear objetos de la clase Persona
persona1 = Persona("Juan", 30)
persona2 = Persona("María", 25)

# Utilizar métodos de la clase Persona
print(persona1.presentarse()) #¡Hola! Mi nombre es Juan y tengo 30 años.
print(persona2.presentarse())#¡Hola! Mi nombre es María y tengo 25 años.


# Clase Rectángulo - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Rectangulo():
    def __init__(self, ancho, altura):
        self.ancho = ancho
        self.altura = altura

    def calcular_area(self):
        return self.ancho * self.altura

# Crear objetos de la clase Rectangulo
rectangulo1 = Rectangulo(5, 10)
rectangulo2 = Rectangulo(3, 7)

# Utilizar métodos de la clase Rectangulo
print("Área del rectángulo 1:", rectangulo1.calcular_area()) # Área del rectángulo 1:
50
print("Área del rectángulo 2:", rectangulo2.calcular_area()) # Área del rectángulo 2:
21


# Clase Libro - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Libro():
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def informacion(self):
        return f"El libro '{self.titulo}' fue escrito por {self.autor}."

# Crear objetos de la clase Libro
libro1 = Libro("Harry Potter", "J.K. Rowling")
libro2 = Libro("Cien años de soledad", "Gabriel García Márquez")

# Utilizar métodos de la clase Libro
print(libro1.informacion())#El libro 'Harry Potter' fue escrito por J.K. Rowling.
print(libro2.informacion())#El libro 'Cien años de soledad' fue escrito por Gabriel García
Márquez.
```

```python
# Método para verificar si un número es par - - - - - - - - - - - - - - - - - - - -
class Numero:
    def __init__(self, valor):
        self.valor = valor

    def es_par(self): # este es un método
        if self.valor % 2 == 0:
            return True
        else:
            return False
# Crear objeto de la clase Numero
numero = Numero(7)
# Llamar al método para verificar si el número es par
if numero.es_par():
    print("El número es par.")
else:
    print("El número es impar.")
```

Ejercicios

```python
# Convertir a objeto - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Usuario():
    def __init__(self, usuario, identificador, grupo):
        self.usuario = usuario
        self.identificador = identificador
        self.grupo = grupo
        if identificador == 1 and grupo == 1:
            self.tipo = 'root'
        else:
            self.tipo = 'normal'
    def __repr__(self):
        return '%s (%s)' %(self.usuario, self.tipo)

if __name__ == '__main__':
    entrada = input()
    partes = entrada.split(':')
    usuario = partes[0]
    identificador = int(partes[1])
    grupo = int(partes[2])
    print(Usuario(usuario, identificador, grupo))
```

```
Entrada: admin:1:1
Salida: admin (root)
```

```python
# De ruta a objeto - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Ruta:
    def __init__(self, ruta):
        if '/' in ruta:
            self.directorio, self.archivo = ruta.rsplit('/', 1)
            self.directorio += '/'
        else:
            self.directorio = './'
            self.archivo = ruta
        self.extension = self.archivo.split('.')[-1]
    def __repr__(self):
        return
f'directorio:{self.directorio}:archivo:{self.archivo}:extension:{self.extension}'
if __name__ == '__main__':
    ruta = input()
```

```
    objeto_ruta = Ruta(ruta)
    print(objeto_ruta)

Entrada: /home/usuario/clases/estructuras/ejercicio.py
Salida: directorio:/home/usuario/clases/estructuras/:archivo:ejercicio.py:extension:py


# Objetos fracción iguales - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Fraccion():
    def __init__(self, numerador, denominador):
        self.numerador = numerador
        self.denominador = denominador

    def __repr__(self):
        return '%s/%s' % (self.numerador, self.denominador)

# lo que implemente para el calculo - - - - - - - - - - - - - - - - -
    def __eq__(self, other):
        return self.numerador * other.denominador == other.numerador * self.denominador

if __name__ == '__main__':
    numerador1 = int(input())
    denominador1 = int(input())
    numerador2 = int(input())
    denominador2 = int(input())

fraccion1 = Fraccion(numerador1, denominador1)
fraccion2 = Fraccion(numerador2, denominador2)
print(fraccion1 == fraccion2)
```

TEMA 5: LISTAS

```
# Crear una lista - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if __name__ == '__main__':
    n_lista = int(input())
    lista = [int(input()) for _ in range(n_lista)]
    print(lista)
```

Ejercicios

```
# Ejercicio 1: Número mayor - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def numMayor(lista):
    mayor = lista[0]
    for num in lista:
        if num > mayor:
            mayor = num
    return mayor

if __name__ == '__main__':
    n_lista = int(input())
    lista = [int(input()) for _ in range(n_lista)]
    mayor = numMayor(lista)
    print(mayor)


# Ejercicio 3: Producto - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def producto(lista):
    resultado = 1
    for num in lista:
        resultado = resultado * num
```

```
    return resultado

if __name__ == '__main__':
    n_elementos = int(input())
    lista = [int(input()) for _ in range (n_elementos)]
    multiplicacion = producto(lista)
    print(multiplicacion)


# Ejercicio 4: Encontrar el elemento más frecuente - - - - - - - - - - - - - - - - - -
def elemento_mas_frecuente(lista):
    mayor = 0
    res = 0
    for i in range (len(lista)):
        if lista.count(lista[i]) > mayor:
            mayor = lista.count(lista[i])
            res = lista[i]
    return res

if __name__ == '__main__':
    n = int(input())
    lista = [int(input()) for _ in range (n)]
    frecuente = elemento_mas_frecuente(lista)
    print(frecuente)


# Interseccion de listas - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def obtener_interseccion(lista1, lista2):
    interseccion = []
    for elemento in lista1:
        if elemento in lista2 and elemento not in interseccion:
            interseccion.append(elemento)
    interseccion.sort()
    return interseccion

N = int(input())
M = int(input())
lista1 = [int(input()) for _ in range(N)]
lista2 = [int(input()) for _ in range(M)]
resultado = obtener_interseccion(lista1, lista2)
print(resultado)


# Eliminar elementos de una lista - - - - - - - - - - - - - - - - - - - - - - - - - -
def eliminarElementos(v_valoreliminado, n_numelem, lista):
    nueva_lista = [num for num in lista if num != v_valoreliminado]
    print(nueva_lista)
if __name__ == "__main__":
    v_valoreliminado = int(input())
    n_numelem = int(input())
    lista = [int(input())for _ in range (n_numelem)]
    eliminarElementos(v_valoreliminado, n_numelem, lista)


# Llenar matriz - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def llenar_matriz (F,C):
    matr=[]
    for _ in range(F):
        matr.append([])
        for i in range (C):
            matr[_].append(int(input()))
```

```python
    return matr
if __name__ == '__main__':
    F = int(input())
    C = int(input())
    print(llenar_matriz(F, C))


# Suma diagonal matriz - - - - - - - - - - - - - - - - - - -
def crear_matriz(F, C):
    matriz = []
    for _ in range(F):
        fila = []
        for _ in range(C):
            valor = int(input())
            fila.append(valor)
        matriz.append(fila)
    return matriz
#- - - - - - - - - - - - - - - - - - - -
def suma_diagonal(matriz):
    suma = 0
    for i in range(len(matriz)):
        suma += matriz[i][i]
    return suma
# - - - - - - - - - - - - - - - - - - -
F = int(input())
C = int(input())
matriz = crear_matriz(F, C)
resultado = suma_diagonal(matriz)
print(resultado)


# Suma columnas - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def crear_matriz(F, C):
    matriz = []
    for _ in range(F):
        fila = [int(input()) for _ in range(C)]
        matriz.append(fila)
    return matriz

def suma_columnas(matriz):
    sumas = [0] * len(matriz[0])
    for fila in matriz:
        for j in range(len(fila)):
            sumas[j] += fila[j]
    return sumas

if __name__ == '__main__':
    F = int(input())
    C = int(input())
    matriz = crear_matriz(F, C)
    resultado = suma_columnas(matriz)
    print(','.join(str(num) for num in resultado))


# Sumar filas - - - - - - - - - - - - - -
def crear_matriz(filas, columnas):
    matriz = []
    for _ in range(filas):
        fila = []
        for _ in range(columnas):
            valor = int(input())
```

```
            fila.append(valor)
        matriz.append(fila)
    return matriz
def sumar_filas(matriz):
    contador = []
    for fila in matriz:
        aux = 0
        for celda in fila:
            aux += celda
        contador.append(aux)
    return contador
if __name__ == '__main__':
    filas = int(input())
    columnas = int(input())
    matriz = crear_matriz(filas, columnas)
    print(sumar_filas(matriz))


# Sacar el centro de la matriz - - - - - - - - - - - - - -
def crear_matriz(filas, columnas):
    matriz = []
    for _ in range(filas):
        fila = []
        for _ in range(columnas):
            valor = int(input())
            fila.append(valor)
        matriz.append(fila)
    return matriz

if __name__ == '__main__':
    filas = int(input())
    columnas = int(input())
    matriz = crear_matriz(filas, columnas)
    mitad = int(filas/2)
    print(matriz[mitad][mitad])
```

TEMA 6: PILAS

```
# Imprimir pila - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Pila():
        def __init__(self):
            self.interna = []
        def push(self, valor):
            self.interna.append(valor)

        def pop(self):
            if not self.interna:
                return None
            return self.interna.pop()
        def peek(self):
            if not self.interna:
                return None
            return self.interna[-1]
        def __repr__(self):
            return str(self.interna)

if __name__ == '__main__':
    longitud_lista = int(input())
    pila = Pila()
    for _ in range (longitud_lista):
```

```
        pila.push (int(input()))
print(pila)
```

TEMA 7: COLAS

```
# Imprimir cola - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Cola():
    def __init__(self):
        self.interna = []
    def esta_vacia(self):
        return not self.interna
    def append(self, valor):
        self.interna.append(valor)
    def shift(self):
        if self.esta_vacia():
            return None
        val = self.interna[0]
        self.interna = self.interna[1:]
        return val
    def peek(self):
        if self.esta_vacia():
            return None
        return self.interna[0]
    def __repr__(self):
        return str(self.interna)
    def copiar(self):
        nueva_cola = Cola()
        for elemento in self.interna:
            nueva_cola.append(elemento)
        return nueva_cola

def ImprimeCola(cola):
    nueva_cola = cola.copiar()
    res = []
    while nueva_cola.peek():
        res.append(str(nueva_cola.shift()))
    return ','.join(res)

if __name__ == '__main__':
    cola = Cola()
    n = int(input())
    for _ in range (n):
        cola.append(int(input()))
print(ImprimeCola(cola))
```

TEMA 9: DICCIONARIOS

```
# imprimir un diccionario - - - - - - - - -
def imprimir(dicc:dict):
    res = sorted(dicc.keys())
    for i in res:
        print (f'{i}: {dicc[i]}')
```

```python
# Crear un diccionario - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def leer_diccionario(elementos: int) -> dict:
    """
    Lee un diccionario desde entrada estándar.
    Las llaves son cadenas y los valores enteros.
    elementos: int
    returns: dict
    """
    res = {}
    for _ in range(elementos):
        llave = input()
        valor = int(input())
        res[llave] = valor
    return res

if __name__ == '__main__':
    n_elementos = int(input())
    diccionario = leer_diccionario(n_elementos)
    print(diccionario)


# Sumar 1 a diccionario - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def leer_diccionario(elementos: int) -> dict:
    """
    Lee un diccionario desde entrada estándar.
    Las llaves son cadenas y los valores enteros.
    elementos: int
    returns: dict
    """
    res = {}
    for _ in range(elementos):
        llave = input()
        valor = int(input())
        res[llave] = valor + 1
    return res

if __name__ == '__main__':
    n_elementos = int(input())
    diccionario = leer_diccionario(n_elementos)
    for llave, valor in sorted(diccionario.items()):
        print('%s:%s' % (llave, valor))


# Ordenar un diccionario - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
d = {'b': 1, 'c': 2, 'a': 3}
for k, v in sorted(d.items()):
    print('llave %s, valor %s' % (k, v))


def recuperar_info(info: str) -> tuple:
    """
    Procesa una línea de información de usuario
    de acuerdo al formato de shadow,
    regresa el nombre de usuario
    y un diccionario con los campos.
    info: str
    returns: usuario, dict, diccionario de campos
    """
    partes = info.split(':')  #Divide la cadena 'info' en una lista usando ':'
    usuario = partes[0]  # Asigna la primera parte de la lista 'partes' a la variable 'usuario'.
    resto = partes[1]  # Asigna la segunda parte de la lista 'partes' a la variable 'resto'.
```

```
    campos = resto.split('$')  # Divide 'resto' en una lista usando '$' como separador.

    dict_campos = {}  # Crea un diccionario vacío 'dict_campos'.
    dict_campos['algoritmo'] = campos[1] #Asigna segundo elem al 'algoritmo' del diccionario.
    dict_campos['salt'] = campos[2]  # Asigna tercer elem de 'campos' al campo 'salt' del diccionario.
    dict_campos['password'] = campos[3]  # Asigna cuarto elem al 'password' del diccionario.
    return usuario, dict_campos  # Devuelve una tupla con 'usuario' y 'dict_campos'.

def construir_diccionario(cadenas: list) -> dict:
    """
    Construye un diccionario de diccionarios
    con la información de las cadenas
    las cadenas usan el formato del archivo
    shadow.
    cadenas: list
    returns: dict
    """
    res = {}  # Diccionario vacío
    for info in cadenas:
        usuario, dict_info = recuperar_info(info)  # Llama a 'recuperar_info' con 'info' y asigna el
resultado
        res[usuario] = dict_info  # Asigna 'dict_info' al diccionario 'res' con la clave 'usuario'.
    return res

if __name__ == '__main__':
    n_usuarios = int(input())  # Lee el num de usuarios y convierte a entero.
    usuario = input()  # Lee nombre de usuario.
    campo = input()  # Lee el nombre del campo.

    cadenas = []  # Inicializa una lista vacía 'cadenas'.
    for _ in range(n_usuarios):
        cadenas.append(input())  # Añade cada línea de información de usuario leída a la lista 'cadenas'.

    informacion = construir_diccionario(cadenas)  # Se asigna el resultado a 'informacion'.
    print(informacion[usuario][campo])  # Imprime valor del 'campo' para el 'usuario' especificado en el
diccionario 'informacion'.



# Ejercicio pasado de diccionarios - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def llenar_diccionario(n):
    diccionario = {}
    for i in range(n):
        linea = input() #lee las partes
        partes = linea.split(':') #['uv.mx', 'OK']
        pagina = partes[0]
        resultado = partes[1] # si es OK o Error
        if not pagina in diccionario and resultado == 'OK':
            diccionario[pagina] = 1 #se crea la llave con el valor de 1
        elif pagina in diccionario and resultado == 'OK':
            diccionario[pagina] += 1 # es más de una vez
    return diccionario

def imprimir_diccionario(diccionario: dict) -> None:
    for llave in sorted(diccionario.keys()):
        if diccionario[llave] >= 2:
            print(llave)

if __name__ == '__main__':
    n = int(input())
    resultado = llenar_diccionario(n)
    imprimir_diccionario(resultado)
```

TEMA 10: RECURSIVIDAD

Función recursiva

1. Caso base: es uno o más casos en los que la función no se llama a sí misma y tiene una solución directa
2. Caso recursivo: debe haber uno o más casos en los que la función se llame a sí misma para resolver un subproblema más pequeño o similar al problema original

```
    # Ejemplo 1: Factorial - - - - - - - - - - - - - - - - - - - - - - - - - -
    # Caso Base: Cuando n es igual a 0 o 1, el factorial es 1.
    # Esta es la condición de detención de la recursión.
    #Caso Recursivo: Para valores de n mayores que 1,
    # la función se llama a sí misma con un argumento más pequeño (n - 1)
    # y multiplica el resultado por n.
    def factorial(n: int) -> int:
        """
        Calcula el factorial de un número de forma recursiva.
        n: int
        returns: int
        """
        if n == 0 or n == 1:
            return 1  # Caso base
        else:
            return n * factorial(n - 1)  # Caso recursivo
    # Ejemplo de uso
    print(factorial(int(input())))


# Ejemplo de un ejercicio Iterativo - - - - - - - - - - - - - - - - - - -

def potencia_iterativa(base: int, exponente: int) -> int:
    """
    Calcula la potencia de un número de forma iterativa.
    base: int
    exponente: int
    returns: int
    """
    resultado = 1
    for _ in range(exponente):
        resultado *= base
    return resultado

# Ejemplo de uso
base = int(input())
exponente = int(input())
print(potencia_iterativa(base, exponente))


# Ejemplo recursivo - - - - - - - - - - - - - - - - - - - - - - - - - - - -

def potencia_recursiva(base: int, exponente: int) -> int:
    """
    Calcula la potencia de un número de forma recursiva.
    base: int
    exponente: int
    returns: int
    """
    if exponente == 0:
        return 1
    elif exponente == 1:
        return base
```

```
    else:
        return base * potencia_recursiva(base, exponente - 1)
# Ejemplo de uso
base = int(input())
exponente = int(input())
print(potencia_recursiva(base, exponente))


# Elemento mayor - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def mayor_rec(lista: list, mayor:int) -> int:
    #caso base
    if not lista:
        return mayor
    #caso recursivo
    frente = lista[0]
    resto = lista[1:]
    if frente > mayor:
        mayor = frente
    return mayor_rec(resto, mayor)
if __name__ == '__main__':
    n = int(input())
    lista = [int(input()) for _ in range (n)]
    print(mayor_rec(lista, 0))


# Numeros pares recursivos - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def son_pares_rec(lista: list, nueva: list) -> list:
    #caso base
    if not lista:
        return nueva
    #caso recursivo
    frente = lista[0]
    resto = lista[1:]
    if frente % 2 == 0:
        nueva.append(frente)
    return son_pares_rec(resto, nueva)
# función no recursiva
def pares(lista):
    return son_pares_rec(lista, [])

if __name__ == '__main__':
    n = int(input())
    lista = []
    for i in range(n):
        lista.append(int(input()))
    print(son_pares_rec(lista, []))
```

TEMA 11: ÁRBOLES

```
# Arboles binarios: Encuentra índice mayor - - - - - - - - - - - - - - - - - - - - - - -
class Nodo():
    def __init__(self, indice, valor=None):
        self.izquierda = None
        self.derecha = None
        self.indice = indice
        self.valor = valor

    def __repr__(self) -> str:
        return '%s:%s' % (self.indice,
```

```
                        self.valor)

class Arbol_binario():
    def __init__(self):
        self.raiz = None
    def agregar_nodo(self, indice, valor=None):
        nodo_nuevo = Nodo(indice, valor)
        if not self.raiz:
            self.raiz = nodo_nuevo
            return
        nodo_actual = self.raiz
        while nodo_actual:
            if indice < nodo_actual.indice:
                if not nodo_actual.izquierda:
                    nodo_actual.izquierda = nodo_nuevo
                    return
                nodo_actual = nodo_actual.izquierda
            else: # por derecha
                if not nodo_actual.derecha:
                    nodo_actual.derecha = nodo_nuevo
                    return
                nodo_actual = nodo_actual.derecha

    def regresar_valor(self, indice: int) -> str:
        """
        Regresa el valor en el índice dado.
        None si no existe el índice
        self, indice
        returns: str
        """
        if not self.raiz:
            return None
        actual = self.raiz
        while actual.indice != indice:
            if indice < actual.indice:
                if not actual.izquierda:
                    return None
                actual = actual.izquierda
            else:
                if not actual.derecha:
                    return None
                actual = actual.derecha
        return actual.valor

    def imprimir_arbol_rec(self, nodo, nivel, res):
        espacios = ''
        for i in range(nivel):
            espacios += '| '
        cadena = res[0]
        cadena += espacios + str(nodo.indice) + ':' + str(nodo.valor) + '\n'
        res[0] = cadena
        if nodo.izquierda:
            self.imprimir_arbol_rec(nodo.izquierda,
                                    nivel +1,
                                    res)
        if nodo.derecha:
            self.imprimir_arbol_rec(nodo.derecha,
                                    nivel +1,
                                    res)

    def __repr__(self) -> str:
        res = ['']
```

```
        self.imprimir_arbol_rec(self.raiz, 0, res)
        return res[0]
    def borrar_nodo(self, indice:int) -> None:
        """
        Borra el nodo en el índice dado.
        self, indice:int
        returns: None
        """
        if indice == self.raiz.indice:
            self.raiz = None
            return
        nodo = self.raiz
        while True:
            # Encontré el nodo que quiero borrar por la izquierda
            if nodo.izquierda and nodo.izquierda.indice == indice:
                nodo.izquierda = None
                return
            # Encontré el nodo que quiero borrar por la derecha
            if nodo.derecha and nodo.derecha.indice == indice:
                nodo.derecha = None
                return
            # Todavía no encuentro el nodo, voy a ver si voy por izquierda
            if indice < nodo.indice:
                if not nodo.izquierda:
                    return
                nodo = nodo.izquierda
            # Todavía no encuentro el nodo, voy a ver si voy por derecha
            else:
                if not nodo.derecha:
                    return
                nodo = nodo.derecha

def leer_arbol(n: int) -> Arbol_binario:
    """
    Regresa un árbol de acuerdo a como lo pasa
    el sistema
    """
    arbol = Arbol_binario()
    for _ in range(n):
        partes = input().split(':')
        indice = int(partes[0])
        valor = partes[1]
        arbol.agregar_nodo(indice, valor)
    return arbol
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def indice_mayor_rec(arbol: Arbol_binario, nodo, mayor):
    if nodo.indice > mayor[0]:
        mayor[0] = nodo.indice
    if nodo.derecha: # mientras haya un nodo derecho
        indice_mayor_rec(arbol, nodo.derecha, mayor) #se encuentra el nodo mayor

def indice_mayor(arbol: Arbol_binario):
    res = [0]
    indice_mayor_rec(arbol, arbol.raiz, res)
    return res[0]
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if __name__ == '__main__':
    n = int(input())
    arbol = leer_arbol(n)
    print(indice_mayor(arbol))
    #print(arbol.regresar_valor(v))
```

```python
# Árboles Generales: Nodos hermanos - - - - - - - - - - - - - - - - - - - - - - - -
class Nodo():
    def __init__(self, indice, padre=None):
        self.hijos = []
        self.indice = indice
        self.padre = padre

    def agregar_hijo(self, hijo):
        self.hijos.append(hijo)

    def __repr__(self):
        return 'n:%s' % self.indice


class Arbol():
    """
    Implementación de un árbol general
    que guarda índices
    los índices son únicos
    """
    def __init__(self):
        self.raiz = None

    def regresarNodo_rec(self, indice, actual, resultado):
        """
        Regresa el objeto nodo con el
        índice dado
        """
        if actual.indice == indice:
            resultado.append(actual)
            return # un poco de poda aunque sea

        for hijo in actual.hijos:
            self.regresarNodo_rec(indice, hijo, resultado)


    def regresarNodo(self, indice):
        res = []
        self.regresarNodo_rec(indice, self.raiz, res)
        if not res:
            return None
        return res[0]


    def agregar_hijo(self, indice, indicePadre=None):
        if not self.raiz:
            nuevo = Nodo(indice)
            self.raiz = nuevo
            return True
        padre = self.regresarNodo(indicePadre)
        nuevo = Nodo(indice, padre)
        if not padre:
            return False
        padre.agregar_hijo(nuevo)
        return True

    def es_hoja(self, indice: int) -> bool:
        """
        Determina si un nodo en un índice dado es una hoja.
```

```
        self, indice: int
        returns: bool
        """
        nodo_interes = self.regresarNodo(indice)
        if not nodo_interes:
            return False
        return len(nodo_interes.hijos) == 0

    def regresar_padre(self, indice: int) -> int:
        """
        Regresa el índice del padre del nodo con
        el índice dado.
        En caso de no existir o no tener padre
        se regresa None

        self,
        indice: int, índice del nodo de interés
        returns: int, índice del padre
        """
        nodo_interes = self.regresarNodo(indice)
        if not nodo_interes:
            return None
        return nodo_interes.padre.indice

    def convertir_a_cadena_arbol_rec(self, nodo: Nodo, nivel, res) -> None:
        """
        Hace un recorrido en profundidad y convierte
        a una cadena
        """
        espacios = ''
        for i in range(nivel):
            espacios += '| '
        cadena = res[0]
        cadena += espacios + str(nodo.indice) + '\n'
        res[0] = cadena
        for hijo in nodo.hijos:
            self.convertir_a_cadena_arbol_rec(hijo, nivel + 1, res)


    def __repr__(self) -> str:
        if self.raiz == None:
            return ''
        res = ['']
        self.convertir_a_cadena_arbol_rec(self.raiz,
                                          0, res)
        return res[0].strip()

    def borrar_nodo(self, indice: int) -> None:
        """
        Borra el nodo en el índice dado de ser
        posible
        """
        if indice == self.raiz.indice:
            self.raiz = None
            return

        nodo = self.regresarNodo(indice)
        if not nodo:
            return

        nodo.padre.hijos.remove(nodo)
```

```python
def leer_arbol(nodos: int) -> Arbol:
    """
    Para leer árboles que pasa el sistema.
    Regresa el árbol resultante

    nodos: int
    returns: Arbol
    """
    arbol = Arbol()
    if nodos == 0:
        return arbol

    indice_raiz = int(input())
    arbol.agregar_hijo(indice_raiz)
    for _ in range(nodos - 1):
        nodo, padre = input().split(':')
        nodo = int(nodo)
        padre = int(padre)
        arbol.agregar_hijo(nodo, padre)

    return arbol

def es_ancestro(arbol: Arbol, indiceA: int, indiceB: int) -> bool:
    """
    Dado un objeto Arbol,
    Determina si el nodo en indiceA es ancestro
    de el nodo en el indiceB.

    arbol: Arbol, indiceA: int, indiceB: int
    returns: bool, True si A es ancestro de B
    """

    nodoB = arbol.regresarNodo(indiceB)
    if not nodoB:
        return False

    siguiente = nodoB.padre
    while siguiente != None:
        if siguiente.indice == indiceA:
            return True
        siguiente = siguiente.padre
    return False
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def hermanos(arbol:Arbol, indiceA: Nodo, indiceB: Nodo) -> bool:
    if indiceA == arbol.raiz.indice or indiceB == arbol.raiz.indice:
        return False
    nodoA = arbol.regresarNodo(indiceA)
    nodoB = arbol.regresarNodo(indiceB)
    if nodoA.padre.indice == nodoB.padre.indice:
        return True
    else:
        return False
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if __name__ == '__main__':
    indiceA = int(input())
    indiceB = int(input())
    n = int(input())
    arbol = leer_arbol(n)
    print(hermanos(arbol, indiceA, indiceB))

# Consejos: Manejar el caso de que se pasa la raiz y te pide una propiedad del padre
```

```python
# si necesitas manejar nodos usa el método regresarNodo


# Árboles Generales: Índice mayor - - - - - - - - - - - - - - - - - - - - - - - - - - -
class Nodo():
    def __init__(self, indice):
        self.hijos = []
        self.indice = indice
    def agregar_hijo(self, hijo):
        self.hijos.append(hijo)
    def __repr__(self):
        return 'n:%s' % self.indice

class Arbol():
    """
    Implementación de un árbol general
    que guarda índices
    los índices son únicos
    """
    def __init__(self):
        self.raiz = None
    def regresarNodo_rec(self, indice, actual, resultado):
        """
        Regresa el objeto nodo con el
        índice dado
        """
        if actual.indice == indice:
            resultado.append(actual)
            return # un poco de poda aunque sea

        for hijo in actual.hijos:
            self.regresarNodo_rec(indice, hijo, resultado)

    def regresarNodo(self, indice):
        res = []
        self.regresarNodo_rec(indice, self.raiz, res)
        if not res:
            return None
        return res[0]


    def agregar_hijo(self, indice, indicePadre=None):
        nuevo = Nodo(indice)
        if not self.raiz:
            self.raiz = nuevo
            return True
        padre = self.regresarNodo(indicePadre)
        if not padre:
            return False
        padre.agregar_hijo(nuevo)
        return True
    def es_hoja(self, indice: int) -> bool:
        """
        Determina si un nodo en un índice dado es una hoja.
        self, indice: int
        returns: bool
        """
        nodo_interes = self.regresarNodo(indice)
        if not nodo_interes:
            return False
        return len(nodo_interes.hijos) == 0
```

```python
def leer_arbol(nodos: int) -> Arbol:
    """
    Para leer árboles que pasa el sistema.
    Regresa el árbol resultante
    nodos: int
    returns: Arbol
    """
    arbol = Arbol()
    if nodos == 0:
        return arbol

    indice_raiz = int(input())
    arbol.agregar_hijo(indice_raiz)
    for _ in range(nodos - 1):
        nodo, padre = input().split(':')
        nodo = int(nodo)
        padre = int(padre)
        arbol.agregar_hijo(nodo, padre)
    return arbol
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def mayor_rec(arbol: Arbol, nodo, res):
    if nodo.indice > res[0]:
        res[0] = nodo.indice
    for hijos in nodo.hijos:
        mayor_rec(arbol, hijos, res)
def mayor(arbol: Arbol):
    res = [0]
    mayor_rec(arbol, arbol.raiz, res)
    return res[0]
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
if __name__ == '__main__':
    n = int(input())
    arbol = leer_arbol(n)
    print(mayor(arbol))



# Árbol General: Contar hojas - - - - - - - - - - - - - - - - - - - - - - - - - -
class Nodo():
    def __init__(self, indice, padre=None):
        self.hijos = []
        self.indice = indice
        self.padre = padre

    def agregar_hijo(self, hijo):
        self.hijos.append(hijo)

    def __repr__(self):
        return 'n:%s' % self.indice


class Arbol():
    """
    Implementación de un árbol general
    que guarda índices
    los índices son únicos
    """
    def __init__(self):
        self.raiz = None

    def regresarNodo_rec(self, indice, actual, resultado):
```

```
        """
        Regresa el objeto nodo con el
        índice dado
        """
        if actual.indice == indice:
            resultado.append(actual)
            return # un poco de poda aunque sea

        for hijo in actual.hijos:
            self.regresarNodo_rec(indice, hijo, resultado)


    def regresarNodo(self, indice):
        res = []
        self.regresarNodo_rec(indice, self.raiz, res)
        if not res:
            return None
        return res[0]


    def agregar_hijo(self, indice, indicePadre=None):
        if not self.raiz:
            nuevo = Nodo(indice)
            self.raiz = nuevo
            return True
        padre = self.regresarNodo(indicePadre)
        nuevo = Nodo(indice, padre)
        if not padre:
            return False
        padre.agregar_hijo(nuevo)
        return True

    def es_hoja(self, indice: int) -> bool:
        """
        Determina si un nodo en un índice dado es una hoja.

        self, indice: int
        returns: bool
        """
        nodo_interes = self.regresarNodo(indice)
        if not nodo_interes:
            return False
        return len(nodo_interes.hijos) == 0

    def regresar_padre(self, indice: int) -> int:
        """
        Regresa el índice del padre del nodo con
        el índice dado.
        En caso de no existir o no tener padre
        se regresa None

        self,
        indice: int, índice del nodo de interés
        returns: int, índice del padre
        """
        nodo_interes = self.regresarNodo(indice)
        if not nodo_interes:
            return None
        return nodo_interes.padre.indice

    def convertir_a_cadena_arbol_rec(self, nodo: Nodo, nivel, res) -> None:
        """
```

```
        Hace un recorrido en profundidad y convierte
        a una cadena
        """
        espacios = ''
        for i in range(nivel):
            espacios += '| '
        cadena = res[0]
        cadena += espacios + str(nodo.indice) + '\n'
        res[0] = cadena
        for hijo in nodo.hijos:
            self.convertir_a_cadena_arbol_rec(hijo, nivel + 1, res)


    def __repr__(self) -> str:
        if self.raiz == None:
            return ''
        res = ['']
        self.convertir_a_cadena_arbol_rec(self.raiz,
                                          0, res)
        return res[0].strip()

    def borrar_nodo(self, indice: int) -> None:
        """
        Borra el nodo en el índice dado de ser
        posible
        """
        if indice == self.raiz.indice:
            self.raiz = None
            return

        nodo = self.regresarNodo(indice)
        if not nodo:
            return

        nodo.padre.hijos.remove(nodo)

    def contar_hojas_rec(self,nodo,res):
        if self.es_hoja(nodo.indice)==True:
            res[0]+=1
        for hijo in nodo.hijos:
            self.contar_hojas_rec(hijo, res)

    def contar_hojas(self):
        res=[0]
        self.contar_hojas_rec(self.raiz,res)
        return res[0]


def leer_arbol(nodos: int) -> Arbol:
    """
    Para leer árboles que pasa el sistema.
    Regresa el árbol resultante

    nodos: int
    returns: Arbol
    """
    arbol = Arbol()
    if nodos == 0:
        return arbol

    indice_raiz = int(input())
    arbol.agregar_hijo(indice_raiz)
```

```
    for _ in range(nodos - 1):
        nodo, padre = input().split(':')
        nodo = int(nodo)
        padre = int(padre)
        arbol.agregar_hijo(nodo, padre)

    return arbol


def es_ancestro(arbol: Arbol, indiceA: int, indiceB: int) -> bool:
    """
    Dado un objeto Arbol,
    Determina si el nodo en indiceA es ancestro
    de el nodo en el indiceB.

    arbol: Arbol, indiceA: int, indiceB: int
    returns: bool, True si A es ancestro de B
    """

    nodoB = arbol.regresarNodo(indiceB)
    if not nodoB:
        return False

    siguiente = nodoB.padre
    while siguiente != None:
        if siguiente.indice == indiceA:
            return True
        siguiente = siguiente.padre
    return False
# - - - - - - - - - - - - - - - - - - - - - - - - -
def contar_hojas_rec(arbol:Arbol,nodo,res):
        if arbol.es_hoja(nodo.indice)==True:
            res[0]+=1
        for hijo in nodo.hijos:
            contar_hojas_rec(arbol,hijo, res)

def contar_hojas(arbol:Arbol,indice):
    nodo=arbol.regresarNodo(indice)
    res=[0]
    contar_hojas_rec(arbol,nodo,res)
    return res[0]
# - - - - - - - - - - - - - - - - - - - - - - - - -
if __name__ == '__main__':
    indice=int(input())
    n = int(input())
    arbol = leer_arbol(n)
    print(contar_hojas(arbol,indice))

#Entrada: nodo a partir de cuál buscar
#Entrada: Número de nodos
#Entrada: Raíz
#Salida: No. de hojas
```

OTROS EJERCICIOS . . .

```
# Ejercicio 2: Xor - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
def xor(num_1, num_2):
    if num_1 == 0 and num_2 == 0 or num_1 == 1 and num_2 == 1:
        return False
    else:
```

```
        return True

if __name__ == '__main__':
    num_1 = int(input())
    num_2 = int(input())
    or_logico = xor(num_1, num_2)
    print(or_logico)

# Ejercicio: en minusculas y sin espacios - - - - - - - - - - - - - - - - - - - -
if __name__=='__main__':
    palabra = input()
    en_minusculas = palabra.lower() # mayusculas a minusculas
    sin_espacios = palabra.replace(" ", "") #replaza espacio, por sin espacio
    print(en_minusculas)
    print(sin_espacios)
```