Estructura de Datos

Segundo Parcial

Tema 9: Diccionarios

```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))
```

Ejercicio Información de usuario

```
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'.
```

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

```python
    # 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))
```

Tema 11: Árboles

Ejercicios en Segundo Parcial