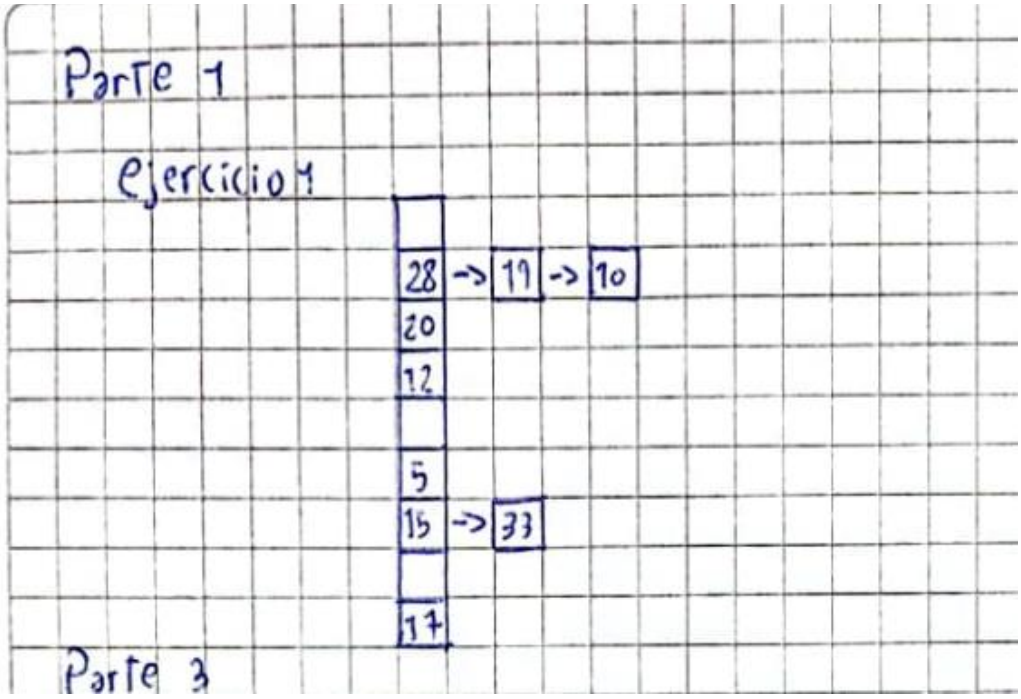
PARTE 1

Ejercicio 1

Ejemplificar que pasa cuando insertamos las llaves 5, 28, 19, 15, 20, 33, 12, 17, 10 en un **HashTable** con la colisión resulta por el método de chaining. Permita que la tabla tenga 9 slots y la función de hash:

$$H(k) = k \bmod 9$$

H(5)=5
H(28)=1
H(19)=1
H(15)=6
H(20)=2
H(33)=6
H(12)=3
H(17)=8
H(10)=1



Ejercicio 2

A partir de una definición de diccionario como la siguiente:

```
dictionary = Array(m,0)
```

Crear un módulo de nombre **`dictionary.py`** que **implemente** las siguientes especificaciones de las operaciones elementales para el **TAD diccionario** .

Nota: puede **`dictionary`** puede ser redefinido para lidiar con las colisiones por encadenamiento

insert(D,key, value)

Descripción: Inserta un key en una posición determinada por la función de hash (1) en el diccionario (dictionary). Resolver colisiones por encadenamiento. En caso de keys duplicados se anexan a la lista.
Entrada: el diccionario sobre el cual se quiere realizar la inserción y el valor del key a insertar
Salida: Devuelve D

dictionary.py > ...

```python
class dictionaryNode:
    value=None
    key=None

def hash_table(m):
    tabla=[None]*m
    return tabla

def insert(D,key,value):
    hash=fun_hash(key,len(D))
    newNode=dictionaryNode()
    newNode.key=key
    newNode.value=value
    if D[hash]==None:
        L=[]
        L.append(newNode)
        D[hash]=L
    else:
        D[hash].append(newNode)
    return D

```

search(D,key)

Descripción: Busca un key en el diccionario
Entrada: El diccionario sobre el cual se quiere realizar la búsqueda (dictionary) y el valor del key a buscar.
Salida: Devuelve el value de la key. Devuelve **None** si el key no se encuentra.

```python
def search(D,key):
    hash=fun_hash(key,len(D))
    if D[hash]==None: return None
    else:
        nodo=buscar_nodo(D[hash],key)
        if nodo==None: return None
        else: return nodo.value


```

delete(D,key)

Descripción: Elimina un key en la posición determinada por la función de hash (1) del diccionario (dictionary)
Poscondición: Se debe marcar como nulo el **key** a eliminar.
Entrada: El diccionario sobre el se quiere realizar la eliminación y el valor del key que se va a eliminar.

Salida: Devuelve **D**

```python
def delete(D,key):
    hash=fun_hash(key,len(D))
    if D[hash]==None: return None
    if len(D[hash])==1:
        if D[hash][0].key==key:
            D[hash]=None
        else: return None
    else:
        nodo=buscar_nodo(D[hash],key)
        if nodo!=None:
            D[hash].remove(nodo)
        else: return None
    return D



def fun_hash(key,m):
    return (key%m)


##busca el nodo de determiana de key en una lista y devuelve el nodo, en caso de no encontrarlo devuelve Nc
def buscar_nodo(L,key):
    if L==None: return None
    for i in range (0,len(L)):
        if L[i].key==key: return L[i]
    return None
```

PARTE 2

Ejercicio 3

Considerar una tabla hash de tamaño m = 1000 y una función de hash correspondiente al método de la multiplicación donde A = (sqrt(5)-1)/2). Calcular las ubicaciones para las claves 61,62,63,64 y 65.

H(61)=700
H(62)=318
H(63)=936
H(64)=554
H(65)=172

Ejercicio 4

Implemente un algoritmo lo más eficiente posible que devuelva **True** o **False** a la siguiente proposición: dado dos strings $s_1...s_k$ y $p_1...p_k$, se quiere encontrar si los caracteres de $p_1...p_k$ corresponden a una permutación de $s_1...s_k$. Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:
Entrada: S = ‘hola’ , P = ‘ahlo’

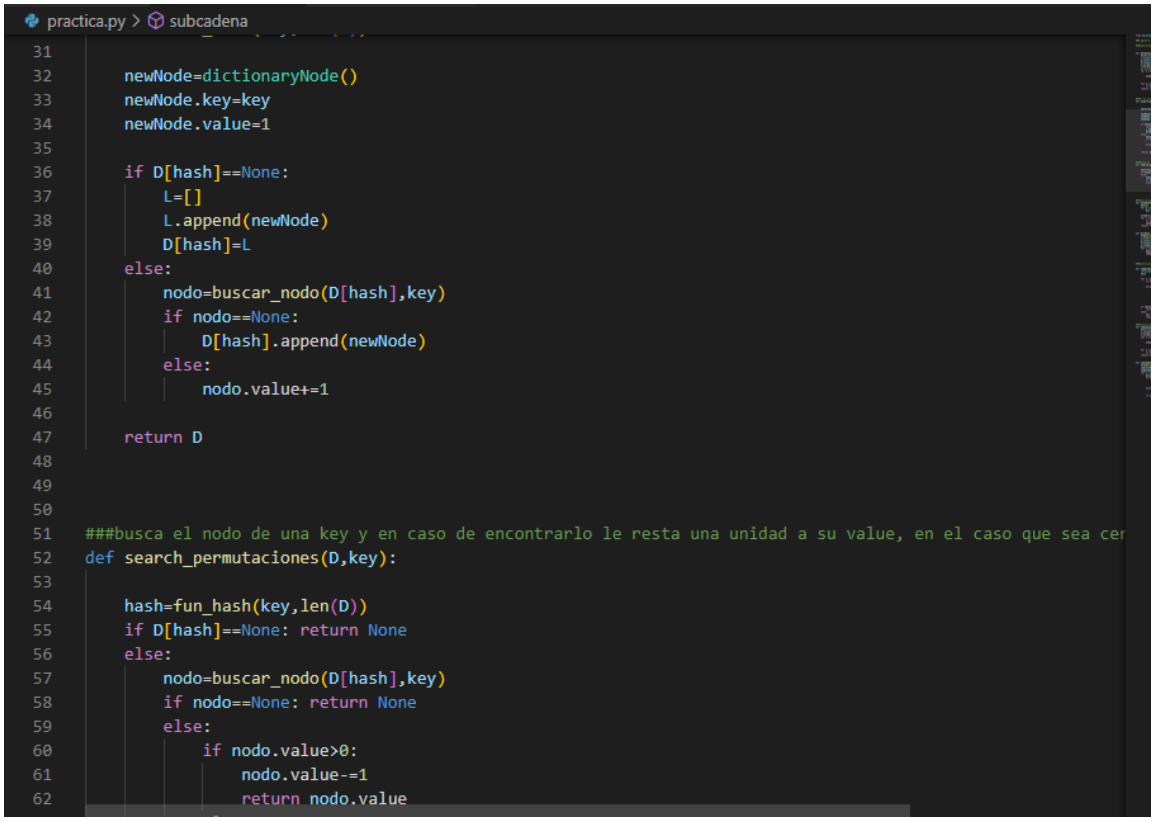
Salida: True, ya que P es una permutación de S

Ejemplo 2:
Entrada: S = ‘hola’ , P = ‘ahdo’
Salida: Falso, ya que P tiene al carácter ‘d’ que no se encuentra en S por lo que no es una permutación de S

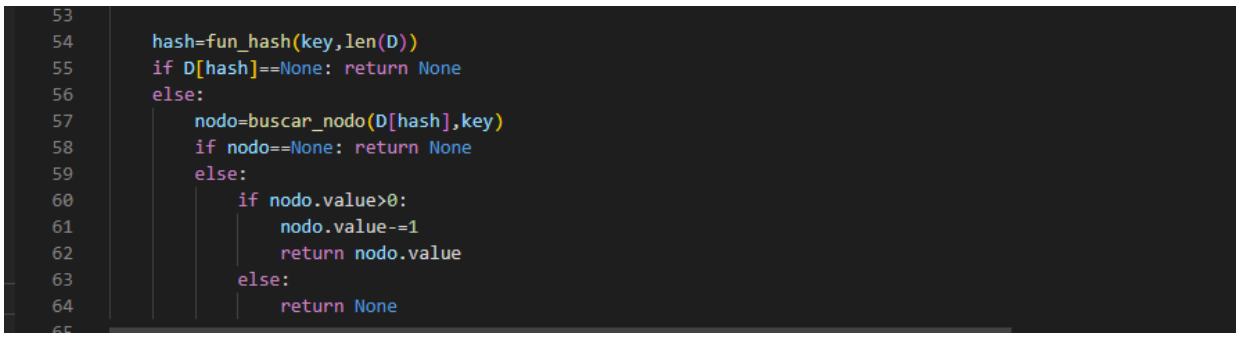
main.py | practica.py | dictionary.py

practica.py > subcadena

```python
from dictionary import*

### parte 2

###ejercicio 4


def permutaciones (S,P):
    D=[None]*27
    s_list=list(S)
    p_list=list(P)
    n_s=len(S)
    n_p=len(P)
    if n_s!=n_p: return False
    for i in range (0,n_s):

        insert_permutaciones(D,ord(s_list[i]))


    for j in range (0,n_p):
        if search_permutaciones(D,ord(p_list[j]))==None: return False
    return True



###es un insert que en el campo value ingresa un una unidad y en caso de ingresar dos key iguales se suman
def insert_permutaciones (D,key):


    hash=fun_hash(key,len(D))

    newNode=dictionaryNode()
```



```
practica.py > subcadena
31
32      newNode=dictionaryNode()
33      newNode.key=key
34      newNode.value=1
35
36      if D[hash]==None:
37          L=[]
38          L.append(newNode)
39          D[hash]=L
40      else:
41          nodo=buscar_nodo(D[hash],key)
42          if nodo==None:
43              D[hash].append(newNode)
44          else:
45              nodo.value+=1
46
47      return D
48
49
50
51  ###busca el nodo de una key y en caso de encontrarlo le resta una unidad a su value, en el caso que sea cer
52  def search_permutaciones(D,key):
53
54      hash=fun_hash(key,len(D))
55      if D[hash]==None: return None
56      else:
57          nodo=buscar_nodo(D[hash],key)
58          if nodo==None: return None
59          else:
60              if nodo.value>0:
61                  nodo.value-=1
62                  return nodo.value
```



```
53
54      hash=fun_hash(key,len(D))
55      if D[hash]==None: return None
56      else:
57          nodo=buscar_nodo(D[hash],key)
58          if nodo==None: return None
59          else:
60              if nodo.value>0:
61                  nodo.value-=1
62                  return nodo.value
63              else:
64                  return None
```

Ejercicio 5

Implemente un algoritmo que devuelva True si la lista que recibe de entrada tiene todos sus elementos únicos, y Falso en caso contrario. Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:
Entrada: L = [1,5,12,1,2]
Salida: Falso, L no tiene todos sus elementos únicos, el 1 se repite en la 1ra y 4ta posición

```python

###ejercicio 5
def elementos_unicos(L):
    dic=[]
    for k in range (0,11):
        dic.append(None)

    n=len(L)
    for i in range(0,n):
        if search(dic,L[i])!=None:return False
        insert(dic,L[i],L[i])
    return True


def primera_ocurrencia(S,P):
    D=hash_table(27)
    s_list=list(S)
    p_list=list(P)
    s_n=len(s_list)
    p_n=len(p_list)
    for i in range (0,s_n):
        key=ord(s_list[i])+ord(s_list[i+p_n])*100
        insert(D,key,i)


```

Ejercicio 6

Los nuevos códigos postales argentinos tienen la forma cddddccc, donde c indica un carácter (A - Z) y d indica un dígito 0, . . . , 9. Por ejemplo, C1024CWN es el código postal que representa a la calle XXXX a la altura 1024 en la Ciudad de Mendoza. Encontrar e implementar una función de hash apropiada para los códigos postales argentinos.

Ejercicio 7

Implemente un algoritmo para realizar la compresión básica de cadenas utilizando el recuento de caracteres repetidos. Por ejemplo, la cadena ‘*aabccccccaaa’* se convertiría en ‘a2blc5a3’. Si la cadena "comprimida" no se vuelve más pequeña que la cadena original, su método debería devolver la cadena original. Puedes asumir que la cadena sólo tiene letras mayúsculas y minúsculas (a - z, A - Z). Justificar el coste en tiempo de la solución propuesta.

```python

###ejercicio 7

def comprension (cadena):
    L=list(cadena)
    G=[]

    for i in range (0,len(L)):
        if i==0:
            G.append(L[0])
        else:
            if L[i]==L[i-1]:
                if G[-1]==L[i-1]:
                    G.append(2)
                else:
                    G[-1]+=1
            else:
                G.append(L[i])
    if len(G)==len(L):
        return cadena
    else:
        ca="".join(map(str,G))
        return ca


```

Ejercicio 8

Se requiere encontrar la primera ocurrencia de un string $p_1...p_k$ en uno más largo $a_1...a_L$. Implementar esta estrategia de la forma más eficiente posible con un costo computacional menor a $O(K*L)$ (solución por fuerza bruta). Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:
Entrada: S = 'abracadabra' , P = 'cada'
Salida: 4, índice de la primera ocurrencia de P dentro de S (abracadabra)

```python
##ejercicio 8
def subcadena (S,T):
    n=len(T)
    indice=0
    comprobante=False
    for i in range(0,n):
        print(T[i])
        if T[i]==S[indice]:
            #print(T[i])
            comprobante=True
            indice+=1
        else:
            comprobante=False
            indice=0
        if indice==len(S):
            return(i-indice+1)


```

Ejercicio 9

Considerar los conjuntos de enteros $S = \{s1, \ldots, sn\}$ y $T = \{t1, \ldots, tm\}$. Implemente un algoritmo que utilice una tabla de hash para determinar si $S \subseteq T$ (S subconjunto de T). ¿Cuál es la complejidad temporal del caso promedio del algoritmo propuesto?

main.py | practica.py | dictionary.py

practica.py > ...

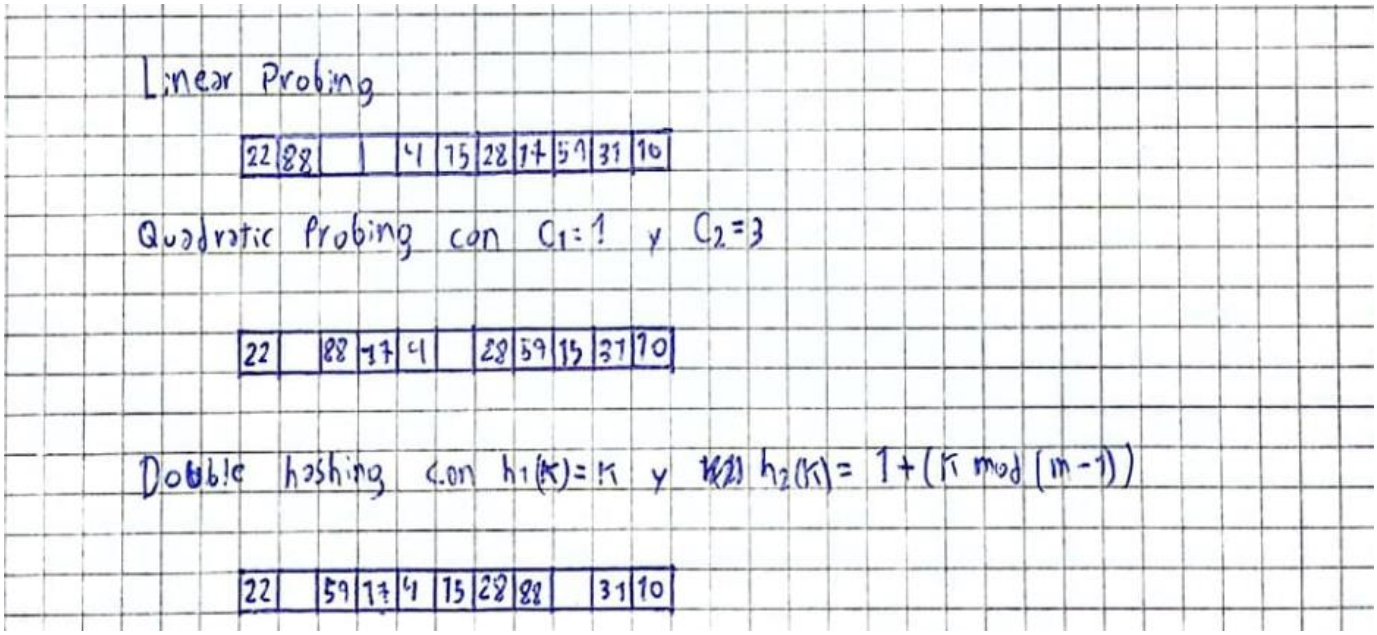
```
115  ###ejercicio 9
116  def subconjunto (S,T):
117      D=[None]*27
118      n_s=len(S)
119      n_t=len(T)
120      for i in range (0,n_t):
121
122          insert_permutaciones(D,T[i])
123
124
125      for j in range (0,n_s):
126          if search_permutaciones(D,S[j])==None: return False
127      return True
128
```

Parte 3

Ejercicio 10

Considerar la inserción de las siguientes llaves: 10; 22; 31; 4; 15; 28; 17; 88; 59 en una tabla hash de longitud m = 11 utilizando direccionamiento abierto con una función de hash *h'(k)* = *k*. Mostrar el resultado de insertar estas llaves utilizando:

1. Linear probing
2. Quadratic probing con *c1 = 1* y c2 = 3
3. Double hashing con *h1(k) = k* y *h2(k) = 1 +(k mod (m - 1))*



Ejercicio 11 (opcional)

Implementar las operaciones de **`insert()`** y **`delete()`** dentro de una tabla hash vinculando todos los nodos libres en una lista. Se asume que un slot de la tabla puede almacenar un indicador (flag), un valor, junto a una o dos referencias (punteros). Todas las operaciones de diccionario y manejo de la lista enlazada deben ejecutarse en O(1). La lista debe estar doblemente enlazada o con una simplemente enlazada alcanza?

Ejercicio 12

Las llaves 12, 18, 13, 2, 3, 23, 5 y 15 se insertan en una tabla hash inicialmente vacía de longitud 10 utilizando direccionamiento abierto con función hash h(k) = k mod 10 y exploración lineal (linear probing). ¿Cuál es la tabla hash resultante? Justifique.

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

La tabla hash resultante es la c ya que en el direccionamiento abierto solo se guarda una key por slot(descartando la opcion D) y se guardan todos los valores aprovechando los slots vacios(descartando A y B).

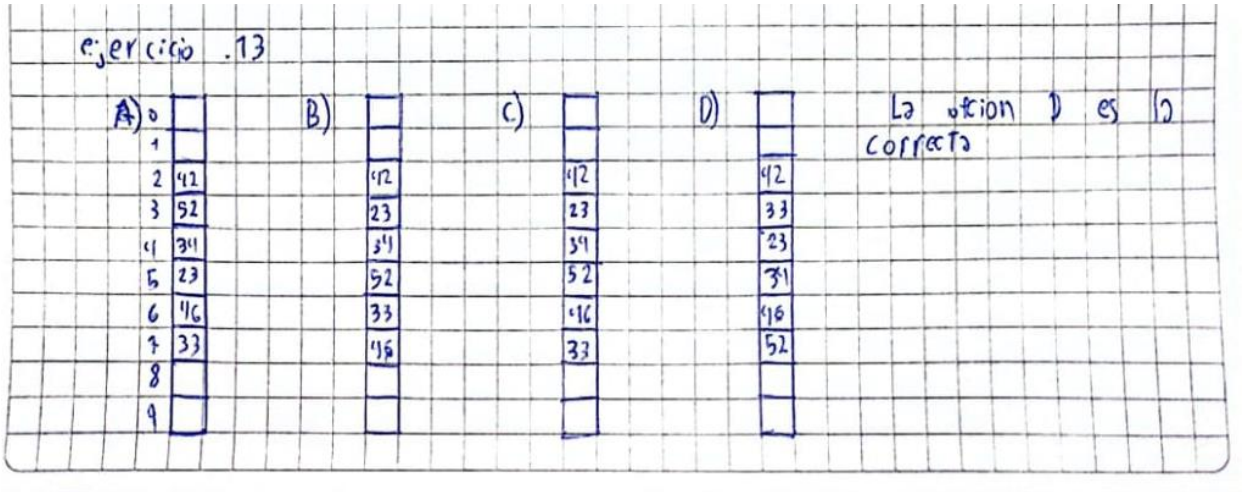
Ejercicio 13

Una tabla hash de longitud 10 utiliza direccionamiento abierto con función hash h(k)=k mod 10, y exploración lineal (linear probing). Después de insertar 6 valores en una tabla hash vacía, la tabla es como se muestra a continuación.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

¿Cuál de las siguientes opciones da un posible orden en el que las llaves podrían haber sido insertadas en la tabla? Justifique

(A) 46, 42, 34, 52, 23, 33
(B) 34, 42, 23, 52, 33, 46
(C) 46, 34, 42, 23, 52, 33
(D) 42, 46, 33, 23, 34, 52



A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca mas allá de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.**~~