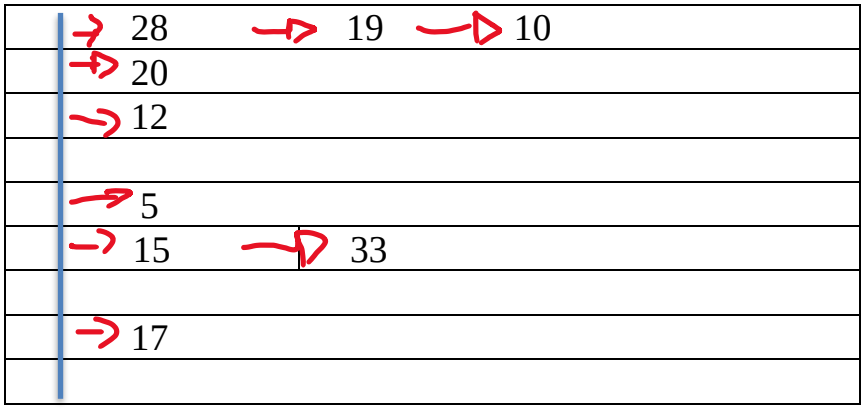
PARTE 1

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:

$$H(k) = k \bmod 9 \qquad (1)$$



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

```
64
65  def insert(m,dic,key,value):
66      pos = key % m #funcion hash (k mod m)
67      if dic[pos] == None: #array en esa pos vacío
68          #creo una lista
69          L = dictionary()
70          addDic(L,key,value)
71          dic[pos] = L
72      else: #encadenamiento
73          addDic(dic[pos],key,value)
74      return dic
75
```

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.

```
81  def search(m,dic,key):
82      pos = key % m #funcion hash (k mod m)
83      if dic[pos] == None:
84          return None
85      else: #busco key
86          current = dic[pos].head
87          while current != None:
88              if current.key == key:
89                  break
90              current = current.nextNode
91          if current == None:
92              return None
93          else:
94              return current.value
```

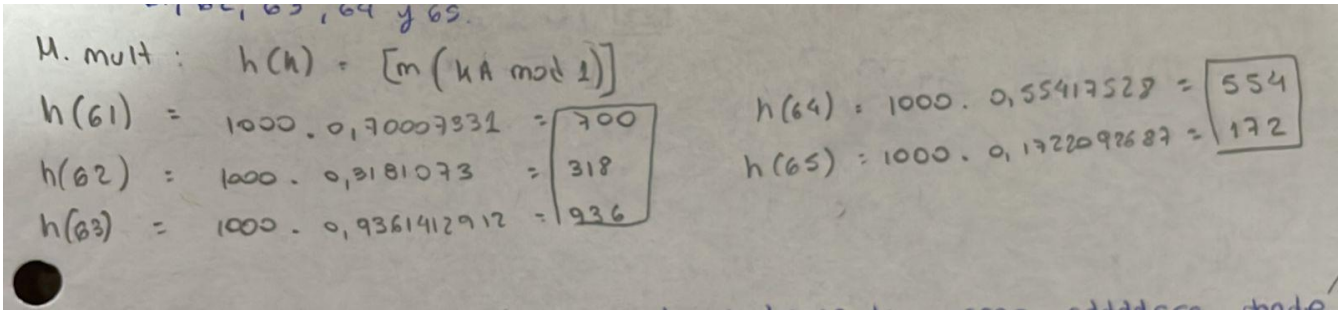
delete(D,key)

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 que se quiere realizar la eliminación y el valor del key que se va a eliminar.
Salida: Devuelve **D**

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.



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

```python
def permutation(s,p):
    if len(p) != len(s):
        return False
    else:
        m = ord("z")-ord("a") #tamaño del abecedario
        dic = [None]*m

        #inserto los caracteres del string p en el diccionario
        for i in range (0,len(p)):
            key = (ord(p[i])-ord("a"))
            insertPosKey(dic, key, 1)

        #"inserto" los caracteres del string s en el diccionario:
        #si encuentro un caracter que ya existe, le resto 1 a value
        #si hay un caracter q no existe, quiere decir que no esta en la otra cadena y no son permutaciones
        for i in range (0,len(p)):
            key = (ord(s[i])-ord("a"))
            permutacion = comparoCaracteres(dic, key, 1)
            if permutacion == False:
                break

        if permutacion == True:
            for i in range (0,len(p)):
                key = (ord(p[i])-ord("a"))
                #reviso que todos los value == 0, caso contrario no es una permutación
                permutacion = esPermutacionFinal(dic, key)
                if permutacion == False:
                    break

        return permutacion
```

El algoritmo planteado es de O(n), siendo n la longitud de las cadenas, ya que recorro la cadena 3 veces de forma individual cada una (no anidadas).

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

```
217 """
218 def elementosUnicos(L):
219     m = len(L)-1
220     A = (math.sqrt(5)-1)/2
221     dic = [None]*m
222     for i in range(0,len(L)): #inserto los elementos en el diccionario
223         current = L[i]
224         if i != 0: #antes de insertar otro elemento me aseguro de que ya no se haya insertado
225             duplicado = search2(A,m,dic,current)
226             if duplicado != None: break #si ya se insertó antes, dejo de agregar elementos al diccionario y devuelvo False
227         insert2(A,m,dic,current,current)
228 
229     if duplicado != None:
230         return False
231     else: return True #si nunca se encontro un elemento duplicaod y se termino con la insercion, devuelvo True
232 
```

La complejidad de este algoritmo es de O(n), siendo n la longitud de la lista, ya que la recorro una sola vez y dentro de ella inserto sus elementos en el diccionario (O(1)) y busco para revisar si hay algún repetido (O(1)).

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.

```
239 
240 def codigoPostal(dic,codigo):
241     nro = (int(codigo[1])+int(codigo[2])+int(codigo[3])+int(codigo[4]))
242     #la key es la suma de los numeros dddd y la suma de las letras en ASCII con peso segun su ubicacion
243     key = (ord(codigo[0])*10^3+ord(codigo[5])*10^2+ord(codigo[5])*10+ord(codigo[5])+nro)
244     m = 10007 #un numero primo grande
245     insertValue_i(dic, key, codigo)
246     return dic
247 
```

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
def comprensionBasica(cadena):
    m = 122         #ord("z")
    #((ord("z")-ord("a")) + (ord("Z")-ord("A"))) #cantidad de posibles caracteres distintos (50)
    #guardo cada caracter en la posicion del numero de su key, es decir que solo en su posicion pueden haber los mismos caracteres
    #en el campo value voy guardando las ocurrencias de cada letra por vez.
    dic = [None]*m
    newCadena = ""
    for i in range(0,len(cadena)):
        if i != 0:
            if cadena[i-1]!=cadena[i]: #si los caracteres cambian, cuento las ocurrencias del anterior y las agrego a la nueva cader
                newCadena1 = cadenaComprimida(cadena[i-1],ord(cadena[i-1]),dic, "")
                newCadena += newCadena1
        key = ord(cadena[i])
        insertPosKey(dic, key, 1)
    if cadena[i-1]==cadena[i]:
        newCadena1 = cadenaComprimida(cadena[i-1],ord(cadena[i-1]),dic, "")
        newCadena += newCadena1
    else:
        newCadena1 = cadenaComprimida(cadena[i],ord(cadena[i]),dic, "")
        newCadena += newCadena1
    if len(newCadena) == len(cadena):
        return cadena
    else:
        return newCadena

def cadenaComprimida(caracter,key,dic,cadenaComprimida): #en esta cadena agrego el caracter y el numero de veces q se repite
    ocurrencia = ocurrencias(dic,key)
    cadenaComprimida += caracter
    cadenaComprimida += str(ocurrencia)
    return cadenaComprimida

def ocurrencias(dic,key): #obtengo la cant de ocurrencias y pongo el valor de esa letra en 0, por si vuelve a aparecer en otro caso
    ocurrencia = dic[key].head.value
    dic[key].head.value = 0
    return ocurrencia
```

La complejidad de este algoritmo es de O(n), siendo n la longitud de la lista ya que esta se recorre una sola vez.

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.

```python
def subcadena(l,c):
    m = (len(l)-len(c)+1)
    final = (len(c))
    cadena=""
    dic = [None]*m
    pos = 0
    for i in range (0,m):
        cadena = l[i:i+final]
        insertValue_i(dic, pos, cadena)
        pos += 1
    indice = searchKey(m,dic,c)
    return indice


def insertValue_i(dic, key, value):        #INSERTO UN ELEMENTO EN EL POSICION pos QUE TENGO EN MI BUCLE FOR
    #creo una lista
    L = dictionary()
    addDic(L,key,value)
    dic[key] = L
    return dic

def searchKey(m,dic,value):                #busca la key a partir de tener como dato el value
    pos = 0                                #la complejidad es O(m) ya que en el peor caso recorre todo el diccionario pero
    #busco value                           #este no tendra encadenamientos
    key = None
    for i in range(0,m):
        if dic[i].head.value == value:
            key = dic[i].head.key
            break
    return key
```

La complejidad de este algoritmo es O(M), siendo M la diferencia entre la longitud de la lista mas larga menos la longitud de la lista más corta

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

```
290  def subcadena(l,c):
291      m = (len(l)-len(c)+1)
292      final = (len(c))
293      cadena=""
294      dic = [None]*m
295      pos = 0
296      for i in range (0,m):
297          cadena = l[i:i+final]
298          insertValue_i(dic, pos, cadena)
299          pos += 1
300      indice = searchKey(m,dic,c)
301      return indice
302
303
304  def insertValue_i(dic, key, value):        #INSERTO UN ELEMENTO EN EL POSICION pos QUE TENGO EN MI BUCLE FOR
305      #creo una lista
306      L = dictionary()
307      addDic(L,key,value)
308      dic[key] = L
309      return dic
310
311  def searchKey(m,dic,value):                #busca la key a partir de tener como dato el value
312      pos = 0                                #la complejidad es O(m) ya que en el peor caso recorre todo el diccionario pero
313      #busco value                           #este no tendra encadenamientos
314      key = None
315      for i in range(0,m):
316          if dic[i].head.value == value:
317              key = dic[i].head.key
318              break
319      return key
320
```

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 ⊆ T (S subconjunto de T). ¿Cuál es la complejidad temporal del caso promedio del algoritmo propuesto?

```
def S_subcjo_T(s,t):

    m = 19 #nro primo
    dic = [None]*m

    for i in range (0,len(t)):
        insert1(m,dic,t[i],t[i])

    for i in range(0,len(s)):
        subcjo = search1(m,dic,s[i])
        if subcjo == None:
            break

    if subcjo == None: return False
    else: return True
```

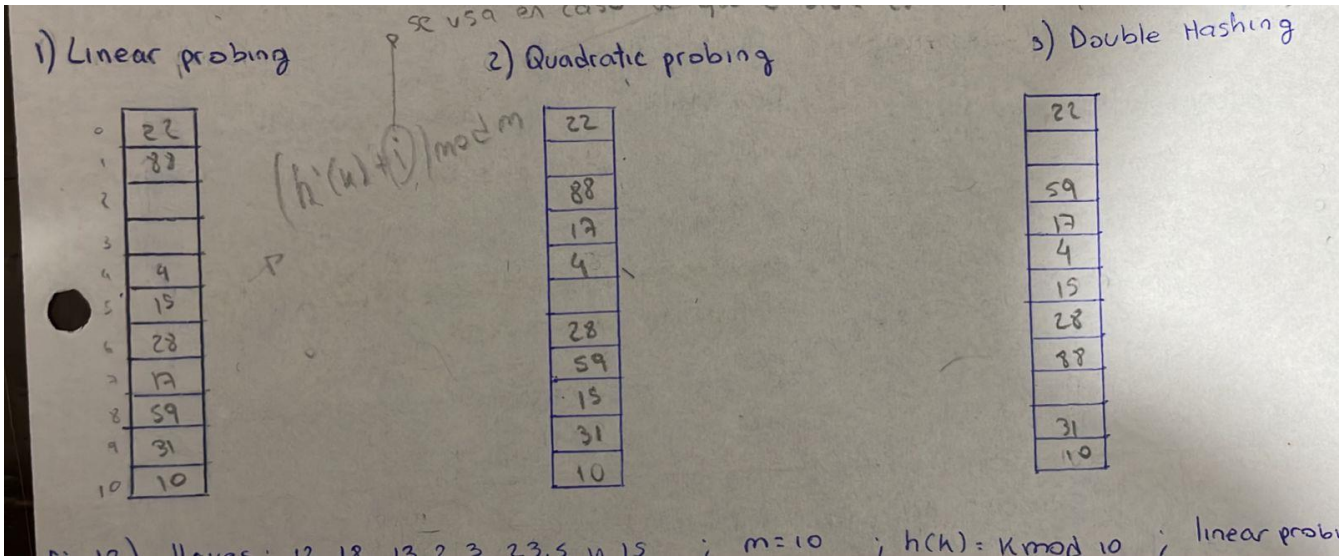
La complejidad en el caso promedio sería de O(s+t) siendo s y t la longitud de los conjuntos.

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	

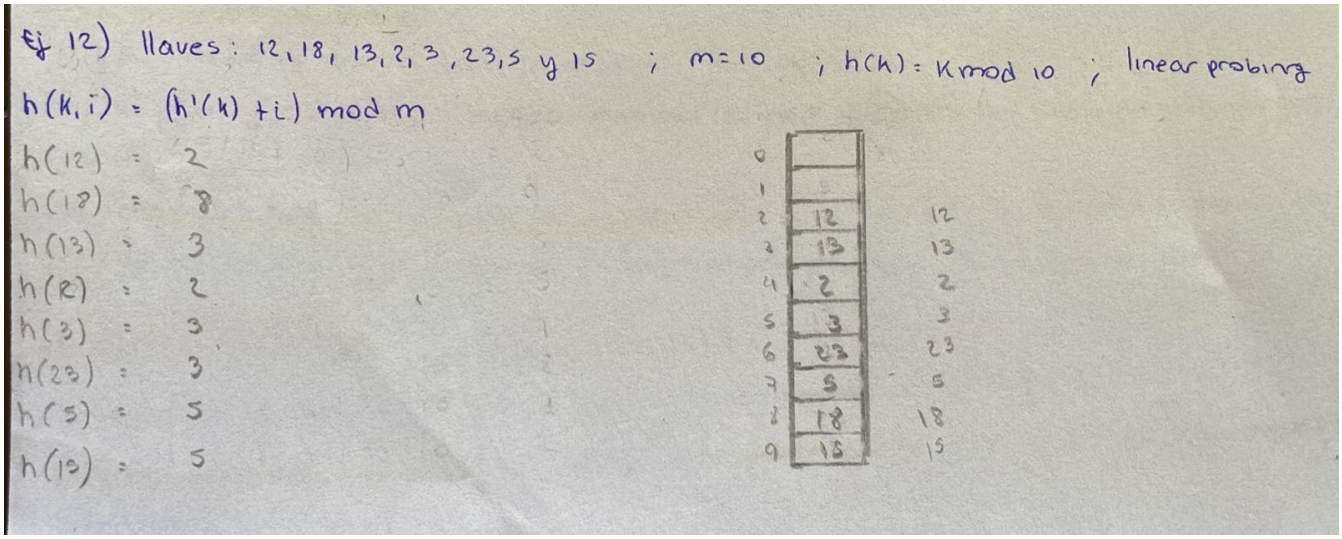
(B)

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)



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.

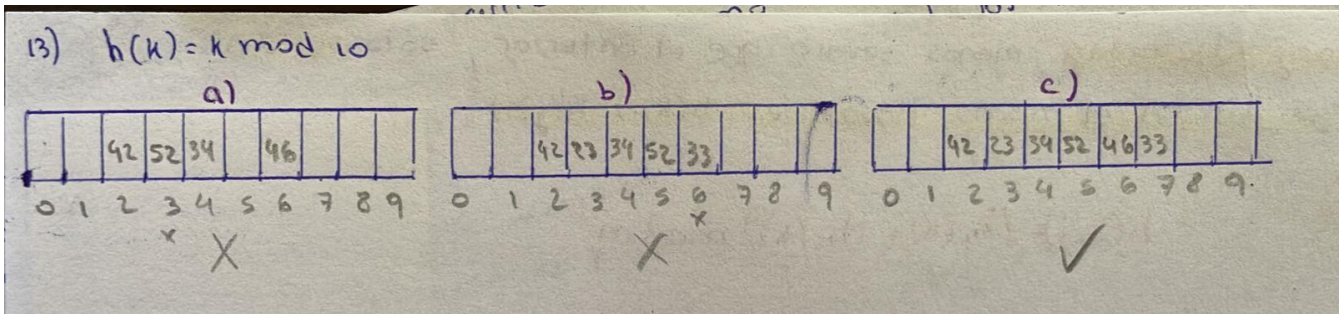
0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

¿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.**~~