Francisco Alba

Ejercicio 1

0			
1	28	19	10
2	20		
3	12		
4			
5	5		
6	15	33	
7			
8	17		

Ejercicio 3

A = (sqrt(5)-1)/2
h(k) = int(1000*(k*A % 1))

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

Ejercicio 10

1. h(k, i) = (k + i) mod 11

22	88			4	15	28	17	59	31	10

2. h(k, i) = (k + i + $3i^2$) mod 11

22		88	17	4		28	59	15	31	10

3. h(k, i) = (k + i(1 + (k mod 10))) mod 11

22		59	17	4	15	28	88		31	10

Ejercicio 12

La tabla de hash resultante es la C, ya que contiene todos los elementos a insertar y los mismos no están encadenados (el direccionamiento abierto con exploración lineal no encadena los elementos).

Ejercicio 13

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

(A)

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

(B)

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

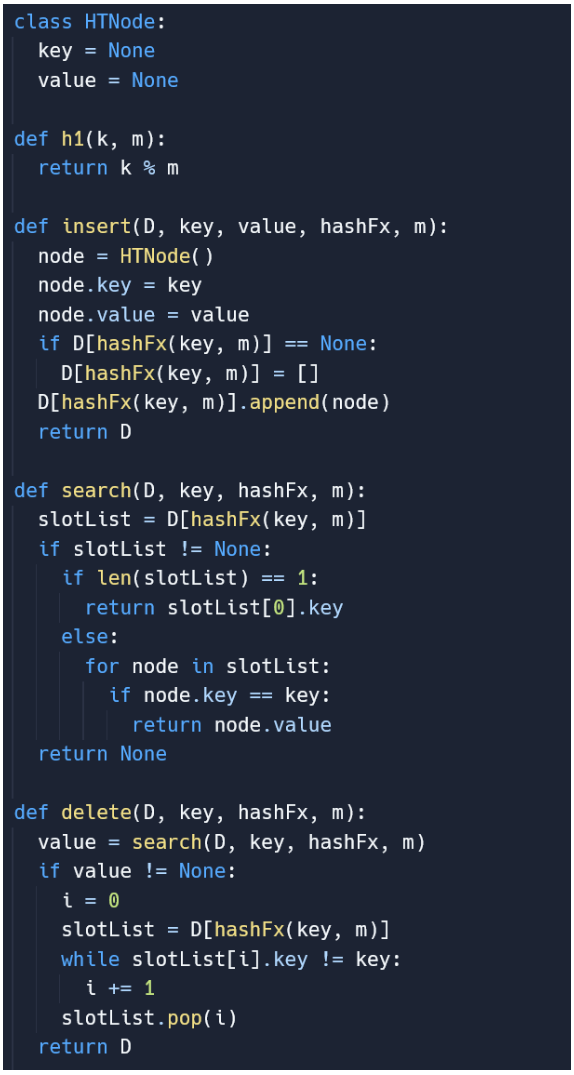
(C)

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

(D)

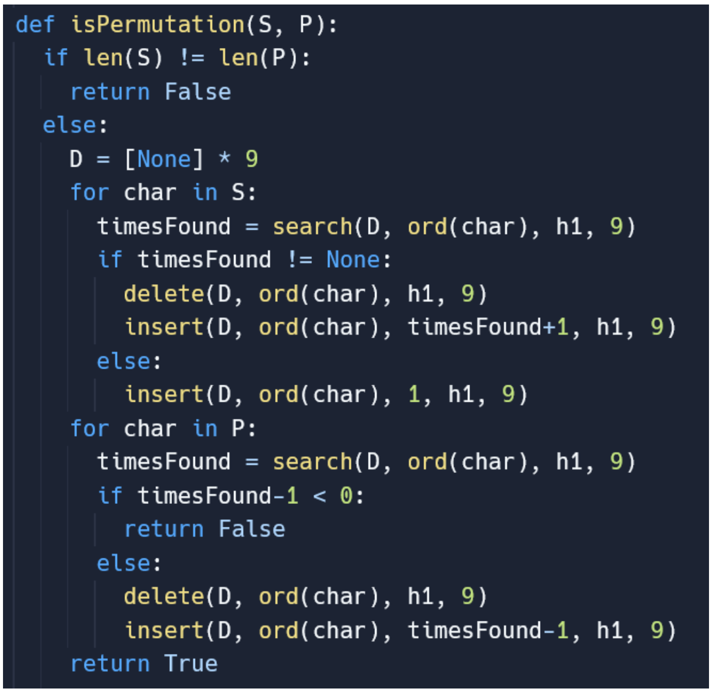
Calculando el resultado de cada posible orden de inserción, es posible observar que la opción C permite llegar al resultado que se muestra como objetivo.

Ejercicio 2



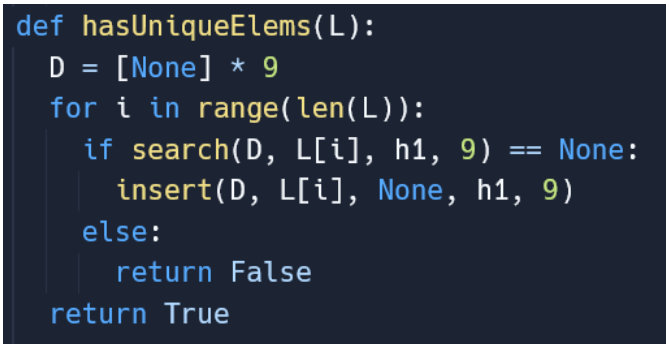
```
class HTNode:
  key = None
  value = None

def h1(k, m):
  return k % m

def insert(D, key, value, hashFx, m):
  node = HTNode()
  node.key = key
  node.value = value
  if D[hashFx(key, m)] == None:
    D[hashFx(key, m)] = []
  D[hashFx(key, m)].append(node)
  return D

def search(D, key, hashFx, m):
  slotList = D[hashFx(key, m)]
  if slotList != None:
    if len(slotList) == 1:
      return slotList[0].key
    else:
      for node in slotList:
        if node.key == key:
          return node.value
  return None

def delete(D, key, hashFx, m):
  value = search(D, key, hashFx, m)
  if value != None:
    i = 0
    slotList = D[hashFx(key, m)]
    while slotList[i].key != key:
      i += 1
    slotList.pop(i)
  return D
```

Ejercicio 4



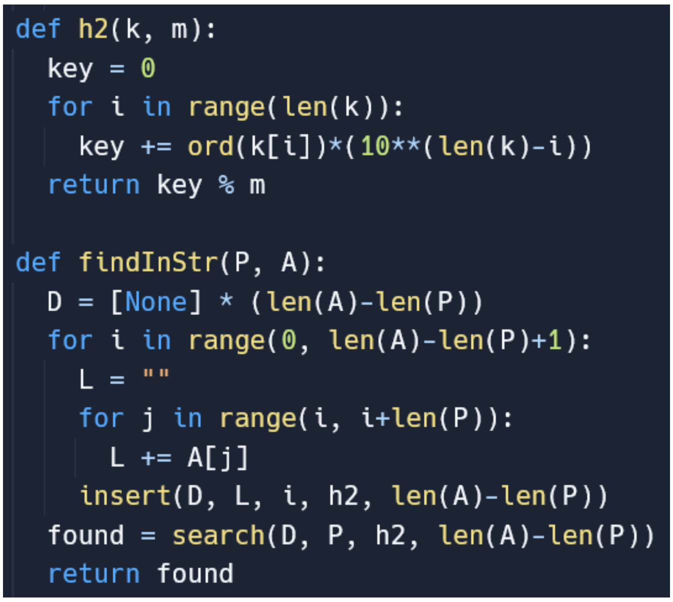
```
def isPermutation(S, P):
  if len(S) != len(P):
    return False
  else:
    D = [None] * 9
    for char in S:
      timesFound = search(D, ord(char), h1, 9)
      if timesFound != None:
        delete(D, ord(char), h1, 9)
        insert(D, ord(char), timesFound+1, h1, 9)
      else:
        insert(D, ord(char), 1, h1, 9)
    for char in P:
      timesFound = search(D, ord(char), h1, 9)
      if timesFound-1 < 0:
        return False
      else:
        delete(D, ord(char), h1, 9)
        insert(D, ord(char), timesFound-1, h1, 9)
    return True
```

Ejercicio 5



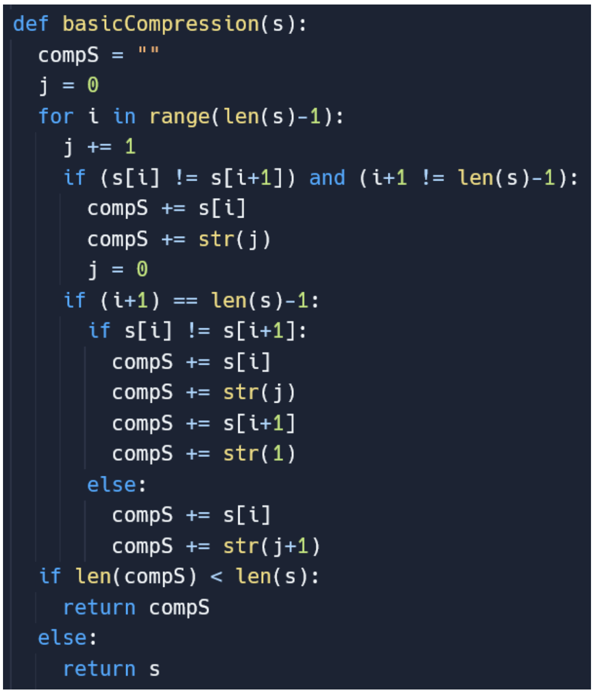
```
def hasUniqueElems(L):
  D = [None] * 9
  for i in range(len(L)):
    if search(D, L[i], h1, 9) == None:
      insert(D, L[i], None, h1, 9)
    else:
      return False
  return True
```

Ejercicio 8



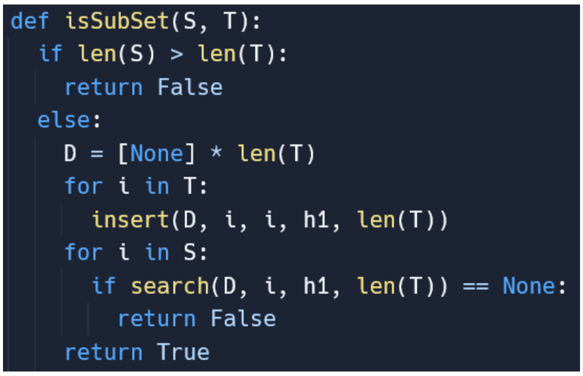
```
def h2(k, m):
  key = 0
  for i in range(len(k)):
    key += ord(k[i])*(10**(len(k)-i))
  return key % m

def findInStr(P, A):
  D = [None] * (len(A)-len(P))
  for i in range(0, len(A)-len(P)+1):
    L = ""
    for j in range(i, i+len(P)):
      L += A[j]
    insert(D, L, i, h2, len(A)-len(P))
  found = search(D, P, h2, len(A)-len(P))
  return found
```

Ejercicio 7



```
def basicCompression(s):
  compS = ""
  j = 0
  for i in range(len(s)-1):
    j += 1
    if (s[i] != s[i+1]) and (i+1 != len(s)-1):
      compS += s[i]
      compS += str(j)
      j = 0
    if (i+1) == len(s)-1:
      if s[i] != s[i+1]:
        compS += s[i]
        compS += str(j)
        compS += s[i+1]
        compS += str(1)
      else:
        compS += s[i]
        compS += str(j+1)
  if len(compS) < len(s):
    return compS
  else:
    return s
```

Ejercicio 9



```
def isSubSet(S, T):
  if len(S) > len(T):
    return False
  else:
    D = [None] * len(T)
    for i in T:
      insert(D, i, i, h1, len(T))
    for i in S:
      if search(D, i, h1, len(T)) == None:
        return False
    return True
```