PARTE 1

Ejercicio 1

[5, 28, 19, 15, 20, 33, 12, 17, 10] $H(k) = k \bmod 9$

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

Ejercicio 2

insert(D,key, value)

```
def insert(D, key, value):
    index = hash_multiplication(key, len(D))
    tuppla = (key,value)
    if D[index] == None:
        D[index] = LinkedList()
    addTupla(D[index], tuppla)
```

search(D,key)

```
def search(D, key):
    index = hash(key, len(D))
    slot = D[index]
    if slot == None or slot.head == None:
        return
    else:
        node = slot.head
        while node != None:
            if node.value[0] == key:
                return key
            node = node.nextNode
        return
```

delete(D,key)

```python
def delete(D, key):
    index = hash(key, len(D))
    slot = D[index]
    if slot == None or slot.head == None:
        return D
    else:
        node = slot.head
        if node == slot.head:
            slot.head = node.nextNode
            return D

        antNode = node
        node = node.nextNode
        while node != None:
            if node.value[0] == key:
                antNode.nextNode = node.nextNode
                return D
            antNode = node
            node = node.nextNode
    return D
```

Funciones Auxiliares:

```python
def hash(k, m):
    return k % m

def hash_multiplication(key, m):
    a=0.6180339887
    frac_part = key * a - int(key * a)
    return int(m * frac_part)


def addTupla(L, t):
    node = Node()
    node.value = t
    if L.head == None:
        L.head = node
    else:
        nodo = L.head
        while nodo.nextNode != None:
            nodo = nodo.nextNode
        nodo.nextNode = node
```

PARTE 2

Ejercicio 3

0	
1	
2	
.	
.	
172	65
.	
.	
318	62
.	
.	
554	64
.	
.	
700	61
.	
.	
936	63
.	
.	
999	

Ejercicio 4

```python
def permutations(S,P):
    weighing = 0
    weighing2 = 0
    for a,b in zip(S,P):
        weighing += ord(a)
        weighing2 += ord(b)

    return weighing == weighing2
```

El orden de complejidad es O(n), ya que recorremos el largo de S y P

Ejercicio 5

```
def singleList(L):
    largo = length(L)
    D = Array(largo, LinkedList())
    node = L.head
    while node != None:
        insert(D, node.value, node.value)
        node = node.nextNode
        if node != None:
            if search(D, node.value) != None:
                return False
    return True
```

El orden de complejidad es O(n), ya que recorremos el largo de la lista, y el algoritmo esta planteado de manera que no exista encadenamiento en el hashtable, por tanto el insert y search quedan siempre en O(1)

Ejercicio 6

```
def hashCorreo(code, m):
    peso = ((ord(code[0]) * 1000) + ((ord(code[5]) * 100)) + ((ord(code[6]) * 10) + (ord(code[7]))))
    number = int(code[1:4])
    return (peso + number) % m
```

Ejercicio 7

```
def compress(string):
    lyric = string[0]
    a = string[0]
    i = 0
    largo = len(string)
    j = 1
    cadena = ""
    while i <= largo:
        if i > 0:
            if a == lyric and i < largo:
                j += 1
            else:
                cadena += a
                if j > 1:
                    cadena += str(j)
                a = lyric
                j = 1
        i += 1
        if i < largo:
            lyric = string[i]
    return cadena
```

El coste temporal es O(n) ya que recorremos la lista, mientras que recorremos la lista, vamos formando la cadena comprimida

Ejercicio 8

```
def idea(P, A):
    lengthLyric = len(P) - 1
    lengthString = len(A) - 1
    sentinel = lengthLyric
    D = Array(1,LinkedList())
    key = 0
    i = 0

    while lengthLyric >= 0:
        key += (ord(P[i]) - ord("a") + 1) * (10 ** lengthLyric)
        lengthLyric -= 1
        i += 1

    insert(D, key, P)

    string = ""
    i = 0
    key = 0
    lengthLyric = sentinel
    j = 0

    while j < lengthString:
        if i <= sentinel:
            key += (ord(A[j]) - ord("a") + 1) * (10 ** lengthLyric)
            string += A[j]
            lengthLyric -= 1
            i += 1
            j += 1
        else:
            if search(D, key) != None:
                return j - i
            string = ""
            key = 0
            i = 0
            lengthLyric = sentinel

    return -1
```

El tiempo de ejecucion es O(n) ya que guardo la key de la palabra a buscar en un hashtable, y despues voy calculando de la misma manera la key pero con la cadena mas larga y la voy comparando con la key que ya guarda. El peor caso seria que la coincidencia este al final de la cadena

Ejercicio 9

```
def subset(S,T):
    lenS = length(S)
    lenT = length(T)
    m = lenT + 1
    if lenS > lenT:
        return False

    D = Array(m, LinkedList())

    node = T.head
    while node != None:
        insert(D, node.value, node.value)
        node = node.nextNode

    node = S.head
    while node != None:
        if search(D, node.value) == None:
            return False
        node = node.nextNode

    return True
```

El orden de complejidad es O(n)

Parte 3

Ejercicio 10

```
def pollProbing():
    numberList = [10,22,31,4,15,28,17,88,59]
    D = Array(11, LinkedList())
    E = Array(11, LinkedList())
    F = Array(11, LinkedList())
    for i in numberList:
        hash_insert(D, i)
    print(D)

    for i in numberList:
        hash_insert2(E, i)
    print(E)

    for i in numberList:
        hash_insert3(F, i)
    print(F)

    print("fin")
```

```python
def hash_insert(D, k):
    i = 0
    while i < len(D):
        j = linearProbingHash(k, len(D), i)
        if D[j] == None:
            D[j] = LinkedList()
            addTupla(D[j], k)
            break
        else:
            i += 1

def hash_insert2(D, k):
    i = 0
    while i < len(D):
        j = quadraticProbingHash(k, len(D), i)
        if D[j] == None:
            D[j] = LinkedList()
            addTupla(D[j], k)
            break
        else:
            i += 1

def hash_insert3(D, k):
    i = 0
    while i < len(D):
        j = doubleHashing(k, len(D), i)
        if D[j] == None:
            D[j] = LinkedList()
            addTupla(D[j], k)
            break
        else:
            i += 1
```

```python
def linearProbingHash(k, m, i):
    return (k + i) % m

def quadraticProbingHash(k, m, i):
    c1 = 1
    c2 = 3
    return (k + c1 * i + c2 * (i ** 2)) % m

def doubleHashing(k,m,i):
    h2 = 1 + (k % (m -1))
    return (k + i * h2) % m
```

a) Linear probing

0	22
1	88
2	
3	
4	4
5	15
6	28
7	17
8	59
9	31
10	10

b)

0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

c)

0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

Ejercicio 12

```
def testLinear():
    numberList = [12, 18, 13, 2, 3, 23, 5, 15]
    D = Array(10, LinkedList())
    for i in numberList:
        j = 1
        node = Node()
        node.value = i
        index = hash(i, 10)
        if D[index] == None:
            D[index] = LinkedList()
            D[index].head = node
        else:
            index += 1
            if D[index] == None:
                D[index] = LinkedList()
                D[index].head = node
            else:
                index += 1
                while D[index] != None:
                    index += 1
                D[index] = LinkedList()
                D[index].head = node
```

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

(C)

Ejercicio 13

```
def testLinear():
    numberList = [46, 34, 42, 23, 52, 33]
    D = Array(10, LinkedList())
    for i in numberList:
        j = 1
        node = Node()
        node.value = i
        index = hash(i, 10)
        if D[index] == None:
            D[index] = LinkedList()
            D[index].head = node
        else:
            index += 1
            if D[index] == None:
                D[index] = LinkedList()
                D[index].head = node
            else:
                index += 1
                while D[index] != None:
                    index += 1
                D[index] = LinkedList()
                D[index].head = node
```

(C) 46, 34, 42, 23, 52, 33