Tablas asociativas en Python: Diccionarios

Fundamentos de Programación

Febrero 2019

Tablas asociativas: ideas generales

- También llamadas *tablas dispersas* o *tablas hash*
- La idea básica es utilizar una clave (en lugar de un entero) como índice para acceder al dato dentro de la estructura
- Internamente, la función de *hashing* (o **función de dispersión**) asocia a cada clave una dirección de acceso en la estructura
- Si la estructura se representa mediante una lista indexada con enteros (o tabla), la función de dispersión asociará a cada clave un índice de la tabla; al menos en teoría, a cada índice le debería corresponder una única clave, de modo que no haya colisiones (dos claves distintas a las que se asocia el mismo índice)
- En la práctica, el tiempo de acceso a un dato en una tabla asociativa será sólo ligeramente superior al tiempo de acceso en una lista indexada con enteros

Tablas asociativas: definiciones

- Función de dispersión: h : U → B
  - U: universo de valores que pueden tomar las claves
  - B = {0, 1, 2, . . . , n-1}: índices de acceso a la tabla
  - n: tamaño de la tabla (n << |U|)

- Colisiones: h(x) = h(y) para x≠y
  - Soluciones:
    1. *Concatenación*: las claves que producen el mismo índice se reúnen en una lista, lo cual mantiene el coste espacial pero incrementa el tiempo medio de búsqueda (internamente, se tendría una lista de listas)
    2. *Direccionamiento abierto*: se define una estrategia (típicamente iterativa) de búsqueda de una ubicación alternativa, o bien, si la tabla está muy llena, se aumenta su tamaño, recalculando los hashes; la primera opción incrementa el tiempo medio de búsqueda, mientras que la segunda incrementa el coste espacial

Tablas asociativas: definiciones

- Factor de carga: L ≡ m/n
  - m: número de datos distintos almacenados en la tabla
  - n: tamaño de la tabla
- En el caso de tablas con direccionamiento abierto: 0 ≤ L ≤ 1
- En el caso de tablas con concatenación, L es la longitud promedio de las listas asociadas a cada índice de la tabla
- El factor de carga óptimo dependerá del balance deseado entre el aprovechamiento de la memoria y el tiempo medio de búsqueda
- Por ejemplo, L entre 0.5 y 1 garantiza tiempos de búsqueda bajos y un aprovechamiento razonable de la memoria; sin embargo, en determinados momentos será necesario realizar costosas operaciones de *concentración* o *redispersión*

Tablas asociativas vs Listas

Acceso en una lista (acceso aleatorio):

- Los elementos de la lista (referencias a datos) son todos del mismo tamaño (S)
- La lista se almacena de forma contigua en memoria
- Para acceder al elemento de índice i se suma a la dirección de comienzo de la estructura un offset = i * S
- Por tanto, el acceso implica un **simple cálculo aritmético**

Acceso en una tabla asociativa (con concatenación):

- Internamente se maneja una lista t de listas
- Para acceder a un elemento indexado por un objeto inmutable x, primero se calcula hash(x) = i (índice de acceso a t)
- Finalmente, se recorre t[i] para buscar un elemento en el que aparezca x
- Así pues, el acceso a una tabla asociativa implica el coste de dos operaciones: **hash(x)** + **búsqueda en una *pequeña* lista**

Tablas asociativas: ventajas

Ventajas:

- Menor coste espacial que una *lista dimensionada por exceso*
- A veces, utilizar tablas asociativas es la única opción:
  - es imposible prever todos los posibles valores de la clave (tal es el caso, por ejemplo, de palabras o frases)
  - el rango completo de valores de la clave es demasiado grande y sobrepasa las capacidades del ordenador

Desventajas (no lo son tanto en la práctica):

- Complejidad teórica y coste temporal de h(x)
- Tiempo medio de acceso mayor que en listas

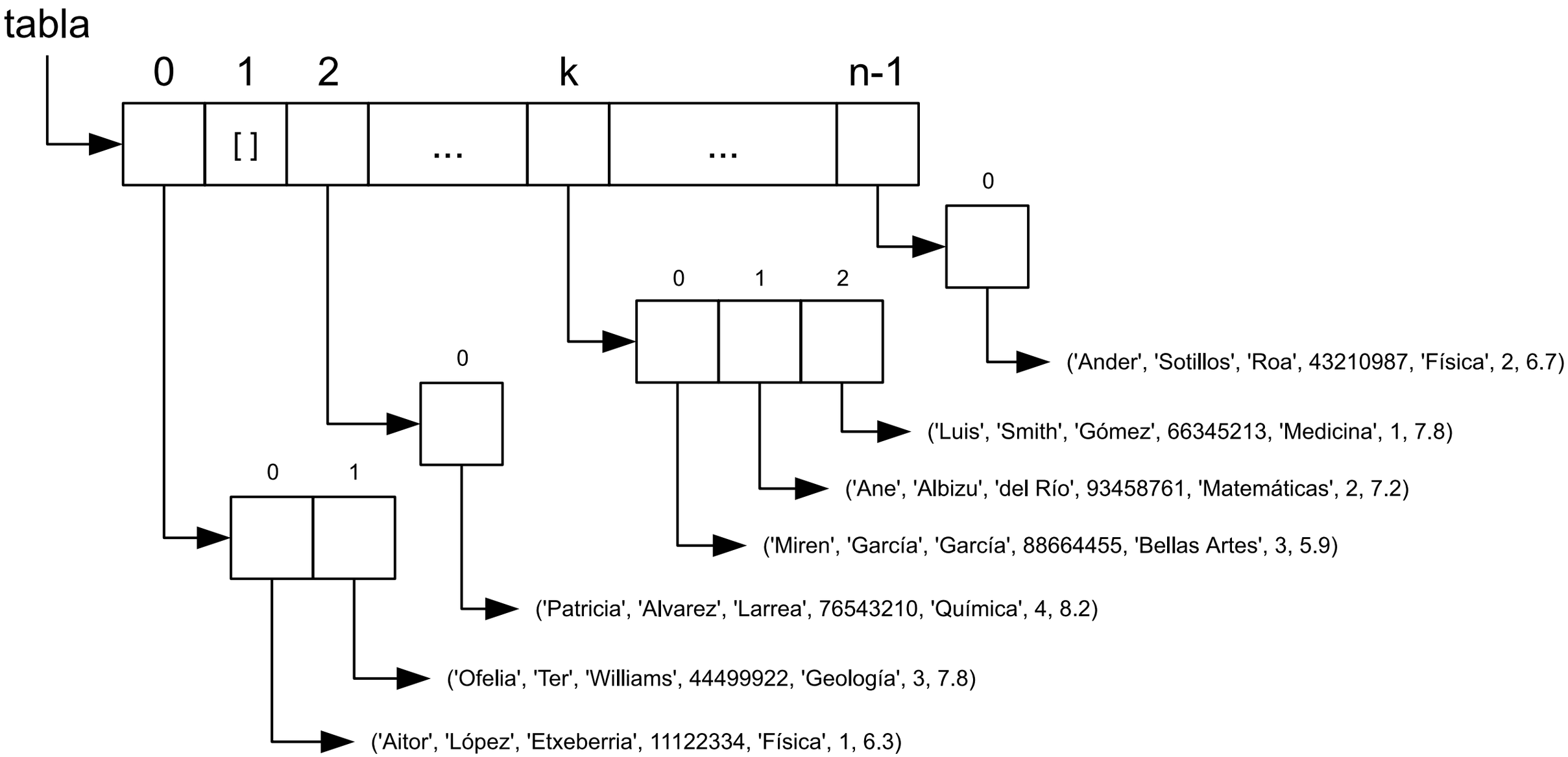
Funciones hash: aplicaciones

- **Tablas asociativas**
- Test de integridad de mensajes
- Comparación de archivos
- Firma digital (autenticación del emisor)
- Almacenamiento y envío de contraseñas
- Actualización de contraseñas (a intervalos fijos)
- Source Code Management (Git)
- Criptografía (PRNG)
- Blockchain (Proof-of-Work, Key derivation)

Tablas asociativas: ejercicio
La tabla de la rectora

- La rectora, debido a restricciones de presupuesto, ya no puede subcontratar ciertos servicios informáticos y nos ha encargado un software en Python para gestionar los datos de los estudiantes.
- Los datos de cada estudiante se depositarán en una tupla, con el nombre, primer apellido, segundo apellido, DNI, grado, curso y nota media.
- Podríamos usar el DNI (un entero) para indexar una lista de fichas. Lamentablemente, el rango de posibles DNIs es mucho mayor que el número de estudiantes matriculados. Si usáramos el DNI como índice entero en una lista, podríamos tener problemas de memoria y además la mayor parte de los elementos de la lista estarían vacíos. Así que conviene usar una tabla asociativa.
- Precisamente, para indexar dicha tabla usaremos como clave el DNI, que permite acceder sin ambigüedad a la información de cada estudiante.
- Internamente, la tabla (de tamaño n) estará representada mediante una lista: el elemento en la posición k de la lista consistirá a su vez en una lista de tuplas (tabla asociativa con concatenación): las de los alumnos a cuyo DNI la función de dispersión ha asignado el índice k.

Tablas asociativas: ejercicio
La tabla de la rectora



Tablas asociativas: ejercicio
La tabla de la rectora

- La rectora nos enviará un fichero de texto con los datos de un/a estudiante en cada línea. Nuestro programa deberá cargar dichos datos en memoria depositándolos en una tabla asociativa, y deberá permitir su gestión posterior (búsqueda, adición/ eliminación de fichas, almacenamiento en archivo, etc.)
- El factor de carga de la tabla deberá estar siempre comprendido en el rango [0.25,4.0] (salvo justo antes de añadir el primer ítem, ya que, lógicamente, si la tabla está vacía el factor de carga será 0).
- El programa deberá incluir las siguientes funciones:
  - una función *cargar(archivo)* que cree una nueva tabla asociativa a partir de los datos almacenados en un archivo de texto, y retorne la tabla resultante.
  - una función *almacenar(tabla,archivo)* que almacene los datos de una tabla asociativa en un archivo, a razón de una tupla por línea.
  - una función *fhash(dni,n)* que tomando como entrada un DNI (entero) devuelva un entero entre 0 y n-1, siendo n el tamaño de la tabla.
  - una función *añadir(datos,tabla)* que añada los datos de un/a estudiante en la posición que corresponda de la tabla (para ello, deberá llamar a la función *fhash()*). La función devolverá el factor de carga resultante.
  - una función *eliminar(dni,tabla)* que elimine de la tabla los datos asociados a un cierto DNI. La función devolverá el factor de carga resultante.

Tablas asociativas: ejercicio
La tabla de la rectora

- Seguimos con la lista de funciones:
  - una función *distribución(tabla)* que cree una nueva tabla de tamaño doble a la tabla original, reubique en ella los datos que estaban en la tabla original, y retorne la tabla resultante. Esta función sólo será necesaria cuando, tras añadir los datos de un/a estudiante, el factor de carga exceda el máximo permitido (4.0).
  - una función *concentración(tabla)* que cree una nueva tabla de tamaño mitad a la tabla original (con un tamaño mínimo de 1), reubique en ella los datos que estaban en la tabla original, y retorne la tabla resultante. Esta función sólo será necesaria cuando, tras eliminar los datos de un/a estudiante, el factor de carga caiga por debajo del mínimo (0.25).
  - una función *retrieve(tupla_patrón,tabla)* que retorne una lista con las tuplas de la tabla asociativa que encajen con la tupla patrón suministrada. La tupla patrón sólo contendrá algunos elementos, el resto serán None y no se utilizarán en la búsqueda. Por ejemplo, si *tupla_patrón = (None, None, None, None, "Física", 3, None)*, la función *retrieve()* debería retornar una lista con todos los estudiantes matriculados en 3º de Física.
  - una función *borrar(tabla)* que elimine todos los elementos de la tabla, dando como resultado una tabla de tamaño 1 vacía.

Diccionarios
Creación e inicialización

- En Python, **un diccionario define una correspondencia (*mapping*) entre claves y valores**: a cada clave le corresponde un valor.
- Internamente, se representa mediante una tabla asociativa (con direccionamiento abierto).
- En la práctica, se maneja como una lista indexada por objetos inmutables: cadenas, tuplas (compuestas a su vez de objetos inmutables), números, etc.
- Aquí nos centraremos en los métodos que Python proporciona para manejar diccionarios: creación, modificación, etc.
- Creación de un diccionario vacío:
  `d={}` o bien `d=dict()`
- Inicialización:
  `d={"mp3": 87.55, "iPad": 566.71, "mouse": 12.75}`
- Asignar valores / Añadir nuevos elementos:
  `d["iPad"]=498.76`

Diccionarios
Funciones y métodos

- **`len(d)`**: devuelve el número de elementos de d
- **`del d[k]`**: elimina el elemento cuya clave es k
- **`d.clear()`**: vacía el diccionario d
- **`k in d`**: devuelve True si k aparece como clave en el diccionario d, y False en caso contrario (nótese que el tiempo búsqueda de una clave en un diccionario es mucho más pequeño que el de un valor en una lista)
- **`d.keys()`**: devuelve una secuencia con las claves del diccionario d
- **`d.values()`**: devuelve una secuencia con los valores almacenados en el diccionario d
- **`d.items()`**: devuelve una secuencia de tuplas (clave,valor) construida a partir de los elementos del diccionario d

Diccionarios
Funciones y métodos

- **IMPORTANTE**: los métodos d.keys(), d.values() y d.items() producen secuencias de tipo dict_keys, dict_values y dict_items, respectivamente. Estas secuencias no son listas y, por tanto, no se les puede aplicar los métodos asociados a listas. Básicamente, lo único que puede hacerse sobre esas secuencias es iterar.
- ¿Qué hacer, entonces, para almacenarlos como listas?
  - **`claves=list(d.keys())`**: crea una lista con las claves de d
  - **`valores=list(d.values())`**: crea una lista con los valores de d
  - **`items=list(d.items())`**: crea una lista de tuplas con los elementos de d
- **`d.copy()`**: devuelve una copia del diccionario d
- **`dict.fromkeys(seq [,value])`**: devuelve un nuevo diccionario tomando como claves los elementos (inmutables) de la secuencia *seq* e inicializando los valores con *value* (por defecto, *None*)

Diccionarios
Funciones y métodos

- **`d1.update(d2)`**: añade a d1 todos los elementos de d2
- **`d.get(k [,value])`**: si existe la clave k en d, devuelve d[k]; si no, devuelve *value* (por defecto, *value*=None)
- **`d.setdefault(k [,value])`**: si existe la clave k en d, devuelve d[k]; si no, devuelve *value* y hace d[k]=*value* (por defecto, *value*=None)
- **`d.pop(k [,default])`**: si la clave k existe en d, devuelve el valor d[k] y elimina el elemento correspondiente de d; en caso contrario, devuelve *default*; si no se suministra un valor por defecto, genera una excepcion de tipo *KeyError*
- **`d.popitem()`**: elimina de d un elemento (clave,valor) al azar y lo devuelve en forma de tupla

Diccionarios
El módulo shelve

El módulo `shelve` de Python permite manejar una estructura de tipo diccionario que no se deposita en memoria, sino que se va almacenando de forma transparente en un fichero, con lo cual no hay limitación de espacio y la estructura puede recuperarse después

La única **limitación** es que **las claves han de ser cadenas de caracteres**

Internamente, el módulo `shelve` usa el módulo `pickle`. Por otra parte, los accesos son calculados, es decir, la **modificación y/o recuperación** de información no requiere recorrer la estructura y, por lo tanto, es **muy rápida**

Veamos cómo funciona:

```
import shelve
a = shelve.open("agenda.shv")
a["Mikel"] = [444556677, "Mikel Lopez", "m.lop@gnail.com"]
a["Laura"] = [444667755, "Laura Aguirre", "l.agui@gnail.com"]
a.close()
```

La extensión `.shv` no es necesaria. El módulo `shelve` crea un fichero de datos binario, codificado y decodificado de forma transparente.

El método `shelve.open()` devuelve un objeto de la clase `shelf` que permite leer y escribir en el fichero mediante un repertorio de métodos de nombre y funcionalidad idénticos a los de un diccionario: `del`, `in`, `clear`, `pop`, etc.

Ventaja importante: `shelve` no carga la estructura en memoria

Diccionarios - Ejemplos de uso
Histograma de caracteres

- Dada una cadena de caracteres, nos interesa conocer el número de apariciones de cada carácter (lo que llamamos *histograma*)
- Podemos crear una secuencia de contadores, de modo que cada vez que vemos un carácter incrementamos el contador correspondiente (usando la función `ord()`)...
- ... o bien podemos crear un diccionario, tomando los caracteres como claves y las cuentas como valores
- La ventaja de usar un diccionario es que no es necesario conocer a priori el número de claves potenciales (ni cuáles son): un diccionario nos permite almacenar información *sólo para aquellas claves que sí aparecen*
- Podemos imitar este comportamiento mediante una lista, pero los mismos cálculos llevan más tiempo, debido a que es necesario localizar los caracteres dentro de la lista (no hay acceso directo)

Diccionarios - Ejemplos de uso
Histograma de caracteres

```
# (1) Secuencia de contadores

MAX_CHAR_CODE=256

def histogram_cnt(s):
    cnt=[]
    for i in range(MAX_CHAR_CODE):
        cnt.insert(i,0)
    for c in s:
        cnt[ord(c)]+=1
    return cnt

# (2) Diccionario

def histogram_d(s):
    d=dict()
    for c in s:
        d[c]=d.get(c,0)+1
    return d
```

```
# (3) Lista (parecido a usar un
#     diccionario… pero más lento)

def histogram_l(s):
    l=list()
    for c in s:
        i=0
        while i<len(l):
            if l[i][0]==c:
                break
            i=i+1
        if i==len(l):
            l.append((c,1))
        else:
            l[i]=(c,l[i][1]+1)
    return l
```

> programa para ver los tiempos de ejecución <

Diccionarios - Ejemplos de uso
Ordenación de claves

- Las claves de un diccionario no se almacenan en un orden aparente (internamente, los elementos se ordenan según los códigos *hash* que corresponden a las claves, pero esto queda oculto al usuario):

```
def print_dict(d):
    for key in d:
        print(key,h[key])

>>> h = histogram_d("perro")
>>> print_dict(h)
p 1
r 2
e 1
o 1
>>>
```

- Si queremos presentar las claves y los valores de un diccionario en un determinado orden, primero debemos ordenar la secuencia de claves y después recorrerla:

```
def print_dict(d):
    l=list(d.keys())
    l.sort()
    for key in l:
        print(key,h[key])

>>> h = histogram_d("perro")
>>> print_dict(h)
e 1
o 1
p 1
r 2
>>>
```

> descargar programa <

Diccionarios - Ejemplos de uso
Histograma inverso

- Dada una cadena de caracteres, ahora nos interesa conocer qué caracteres aparecen exactamente n veces
- Si disponemos del histograma *directo*, nos veremos obligados a recorrerlo hasta el final para construir una lista con los caracteres cuyo número de apariciones sea n... y esto es tan lento como buscar un carácter en una lista (véase el ejemplo del histograma directo)
- Precisamente, lo que podemos hacer es construir un diccionario inverso (*inv*) de listas de caracteres, utilizando el número de apariciones (*n*) como clave
- Una vez construido el diccionario inverso, tendremos acceso directo a la lista de los caracteres que aparecen *n* veces; si *inv[n]* no existe, significará que no hay caracteres que aparezcan *n* veces
- **IMPORTANTE: los valores de un diccionario pueden ser listas o diccionarios, mientras que las claves han de ser objetos inmutables**

Diccionarios - Ejemplos de uso
Histograma inverso

```
# (1) Aproximación básica

def inv_histo(h):
    inv=dict()
    for c in h:
        n=h[c]
        if n in inv:
            inv[n].append(c)
        else:
            inv[n]=[c]
    return inv

>>> h=histogram_d("perro")
>>> ih=inv_histo(h)
>>> print_dict(ih)
1 ['p', 'e', 'o']
2 ['r']
>>>
```

```
# (2) Aproximación compacta

def inv_histo(h):
    inv=dict()
    for c in h:
      n=h[c]
      inv.setdefault(n,[]).append(c)
    return inv

>>> h=histogram_d("los galos lanzaban vacas")
>>> ih=inv_histo(h)
>>> print_dict(ih)
1 ['c', 'b', 'g', 'v', 'z']
2 ['o', 'n']
3 [' ', 'l', 's']
6 [`a']
>>>
```

> descargar programa <

Diccionarios - Ejemplos de uso
Almacenamiento de resultados intermedios

- Determinados problemas admiten una solución recursiva que, aunque sencilla, puede resultar muy ineficiente, debido a que **ciertos resultados se calculan muchas veces**. Por ejemplo, la serie de Fibonacci:

```
def fib(n):     # versión recursiva básica
      if n<2:
          return n
      else:
          return fib(n-1)+fib(n-2)
```

- En este ejemplo, para calcular fib(4), el cálculo de fib(1) se realiza 3 veces, y el de fib(0) y fib(2), 2 veces. El problema de sobre-cómputo es mucho mayor cuanto mayor es n. De hecho, el coste temporal crece exponencialmente con n.
- Una forma de resolverlo consiste en **definir un diccionario fib_d que mantenga los valores ya calculados de fib()**. Antes de calcular fib(k), se consulta el diccionario; si existe fib_d[k], simplemente se retorna su valor; si no, se calcula y se crea un nuevo elemento fib_d[k]. De esta forma, se consume un mayor espacio de memoria pero **cada valor se calcula sólo una vez**.

> descargar programa para medir tiempos con todas las versiones <

Diccionarios - Ejemplos de uso
Almacenamiento de resultados intermedios

- Veamos cómo queda la función:

```
fib_d={0:0, 1:1}

def fib(n):  # versión con memoria
    if not n in fib_d:
        fib_d[n]=fib(n-1)+fib(n-2)
    return fib_d[n]
```

- Estamos usando una variable global de tipo diccionario para almacenar los elementos de la serie de Fibonacci. Esta variable ocupa espacio en memoria y permanece tras ejecutar la función.
- Variables globales de tipo diccionario o lista se pueden modificar (añadir o eliminar elementos) sin necesidad de hacer nada especial. Sin embargo, no se pueden reasignar. Para hacerlo, habría que declararlas como globales.

- Esto es algo general: para reasignar una variable global dentro de una función hay que declararla como tal:

```
ncalls=0
def f():
    global ncalls
    ncalls=ncalls+1
```

- La **implementación óptima de fib()** utiliza sólo la memoria auxiliar necesaria y es aún más rápida:

```
def fib(n):   # versión óptima
    if n<2:
        return n
    x2=0
    x1=1
    for i in range(2,n+1):
        x=x1+x2
        x2=x1
        x1=x
    return x
```

Diccionarios: Ejercicios propuestos (1)

1. Escribir en Python una función que tome como entrada un fichero de texto y retorne el histograma de palabras de dicho fichero, en forma de diccionario. Escríbase asimismo una función que, tomando como entrada ese histograma y un entero n, devuelva la lista de palabras que aparecen exactamente n veces en el fichero (sin usar un histograma inverso). Por último, escríbase un programa que pida al usuario el nombre de un fichero y un valor n, y a continuación imprima en líneas sucesivas las palabras que aparecen n veces.
2. Escribir en Python una función que tome como entrada un fichero de texto y retorne un diccionario con el histograma inverso de palabras de dicho fichero: cada clave será un entero n, al que le corresponderá la lista de palabras que aparecen exactamente n veces en el fichero.
3. Escribir en Python una función que tome como entrada un fichero de texto y retorne un diccionario d_pos tal que d_pos[w] contenga la lista de posiciones (i,j) en las que aparece cada palabra w dentro del fichero, siendo i: índice de línea y j: índice de la palabra dentro de la línea (ambas empezando desde 1). Recuérdese que el final de línea viene marcado por el carácter '\n'.

Diccionarios: Ejercicios propuestos (2)

4. Escribir en Python una función que tome como entrada un entero n>0 y retorne un diccionario f con la descomposición de n en factores primos, tal que las claves del diccionario sean los factores y el valor asociado a cada clave sea el exponente de dicho factor. Así, si n=360, la función deberá retornar el diccionario f = {2: 3, 3: 2, 5: 1}.

5. Repetir el ejercicio anterior usando el módulo `shelve` para, en lugar de retornar la descomposición de n en factores primos, guardarla de manera persistente en un fichero. Además, escribir en Python una función que reciba como entrada dos ficheros `shelve` con la factorización de sendos enteros n y m, y devuelva el máximo común divisor de n y m.

6. Se dice que una matriz es dispersa cuando la mayor parte de sus valores son nulos.
   En Python, una matriz dispersa A puede representarse de manera eficiente mediante un diccionario que contenga únicamente los valores no nulos de la matriz, tal que una tupla de índices (i, j) sirva de clave para acceder al valor almacenado en dicha posición. Así, por ejemplo, la matriz A = [[0,0,3.7,0], [-1.4,0,0,5.21], [0,0,0,1.5], [2.04,1.92,0,0]] se representaría como sigue:

   A_dict = { (0,2): 3.7, (1,0): -1.4, (1,3): 5.21, (2,3): 1.5, (3,0): 2.04, (3,1): 1.92 }.

   Escribir en Python dos funciones, suma(a,b) y producto(a,b), que tomen como entrada dos matrices dispersas a y b (representadas del modo descrito más arriba) y retornen una nueva matriz dispersa con la suma y el producto de ambas, respectivamente.