41-PROGR-diccionario

October 12, 2017

Diccionarios

Disponemos, en "*PALABRAS2.txt*" de una lista de unas 120000 palabras en inglés, y queremos organizar la información en un diccionario de SAGE. Como el uso de este diccionario será para averiguar si una palabra pertenece o no al inglés nos conviene usar como claves los posibles grupos de tres letras iniciales y valores listas de palabras que comienzan por esas tres letras de la clave.

```
In [1]: def diccionario():
            dicc = {}
            infile = open("PALABRAS2.txt","r")
            for palabra in infile.readlines():
                if dicc.has_key(palabra[:3]):
                    dicc[palabra[:3]].append(palabra[:-1])
                else:
                    dicc[palabra[:3]]=[palabra[:-1]]

            return dicc
```

```
In [2]: dicc = diccionario()
```

```
In [4]: print dicc["GEO"]
```

```
['GEOCENTRIC', 'GEOCENTRICALLY', 'GEOCENTRICISM', 'GEOCHEMICAL', 'GEOCHEMICALLY', 'GEOCHEMIST'
```

```
In [5]: print dicc["MAT"]
```

```
['MAT', "MAT'S", 'MATABELE', 'MATABELELAND', 'MATADI', 'MATADOR', 'MATAMOROS', 'MATANZAS', 'MA
```

```
In [6]: def comprueba(C):
            if C in dicc[C[:3]]:
                return True
            else:
                return False
```

```
In [7]: time comprueba('GEOMETRY')
```

```
CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 13.1 µs
```

```
Out[7]: True

In [8]: time comprueba('ABCDEFG')

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 11 µs


Out[8]: False
```

Usando listas

```
In [9]: def listado():
            L = []
            infile = open("PALABRAS2.txt","r")
            for palabra in infile.readlines():
                L.append(palabra[:-1])
            return L

In [10]: L = listado()

In [11]: len(L)

Out[11]: 124341

In [12]: L[100]

Out[12]: 'ABDOMENS'

In [13]: def comprueba_L(C):
             if C in L:
                 return True
             else:
                 return False

In [14]: time comprueba_L('GEOMETRY')

CPU times: user 0 ns, sys: 0 ns, total: 0 ns
Wall time: 708 µs


Out[14]: True

In [15]: time comprueba_L('ABCDEFG')

CPU times: user 1.7 ms, sys: 117 µs, total: 1.82 ms
Wall time: 1.89 ms


Out[15]: False
```

Los tiempos que obtenemos son menores para el diccionario, pero conviene aplicar ambos métodos a un texto corto:

```
In [16]: def comprueba_texto(C):
             cont = 0
             L = C.split()
             for palabra in L:
                 if comprueba(palabra) == True:
                     cont += 1
             return cont

In [17]: texto = "THROUGH THE USE  ABSTRACTION AND LOGICAL REASONING MATHEMATICS DEVELOPED FRO

In [18]: time comprueba_texto(texto)

Out[18]: 72
         Time: CPU 0.00 s, Wall: 0.00 s

In [18]: def comprueba_texto2(C):
             cont = 0
             L = C.split()
             for palabra in L:
                 if comprueba_L(palabra) == True:
                     cont += 1
             return cont

In [19]: time comprueba_texto2(texto)

CPU times: user 69 ms, sys: 1.3 ms, total: 70.3 ms
Wall time: 182 ms


Out[19]: 72
```

Usando un diccionario seguimos obteniendo resultados mejores. Por supuesto, en una situación real, con textos mucho más largos, la diferencia en tiempos debe ser grande.

Versión abstracta

Generamos una lista de 10^6 enteros aleatorios en el intervalo $[100, 10^7]$ que convertimos en cadenas de caracteres (palabras). Enteros aleatorios significa que todos los enteros del intervalo tienen,a priori, la misma probabilidad de aparecer en la lista L.

```
In [20]: L = [str(randint(100,10^7)) for muda in srange(10^6)]
```

Con listas

```
In [21]: def comprobador(L):
             cont = 0
             for muda in srange(10^3):
                 if muda%100 == 0:
                     print "Van otros 100"
                 if str(randint(1,10^7)) in L:
                     cont += 1
             return cont
```

```
In [22]: time comprobador(L)

Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
CPU times: user 16 s, sys: 465 ţs, total: 16 s
Wall time: 26 s


Out[22]: 97
```

Con diccionarios

```
In [23]: def diccionario2():
             dicc = {}
             for palabra in L:
                 if dicc.has_key(palabra[:3]):
                     dicc[palabra[:3]].append(palabra[:-1])
                 else:
                     dicc[palabra[:3]]=[palabra[:-1]]

             return dicc

In [24]: dicc2 = diccionario2()

In [25]: def comprueba2(C):
             if C in dicc2[C[:3]]:
                 return True
             else:
                 return False

In [26]: def comprobador2():
             cont = 0
             for muda in srange(10^3):
                 if muda%100 == 0:
                     print "Van otros 100"
                 if comprueba2(str(randint(100,10^7))) == True:
                     cont += 1
             return cont

In [27]: time comprobador2()
```

```
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
Van otros 100
CPU times: user 63 ms, sys: 42 ţs, total: 63.1 ms
Wall time: 128 ms
```

```
Out[27]: 49
```

Vemos la ventaja enorme, cuando la cantidad de información a manejar es muy grande, de estructurar la información en un diccionario frente a la versión mucho más amorfa de una lista.

Ésto no debería sorprendernos: el diccionario tiene unas 1000 claves y cada una de ellas tendrá como valor una lista de alrededor de 1000 enteros. Una búsqueda en el diccionario equivale, más o menos, a dos búsquedas en listas de longitud 1000, mientras que una búsqueda en una lista de longitud 10^6 es mucho más costosa porque en el peor caso hay que recorrer casi toda la lista buscando nuestro entero.

```
In [29]: max([len(dicc2[key]) for key in dicc2])
```

```
Out[29]: 1227
```

```
In [30]: min([len(dicc2[key]) for key in dicc2])
```

```
Out[30]: 1004
```