UNIVERSITAT DE BARCELONA

Grau d'Enginyeria Informàtica
Facultat de Matemàtiques

Algorísmica

Hashing i Cerca

Jordi Vitrià

Considerem el problema de buscar un determinat valor en una llista.

- Si la llista no està ordenada, la complexitat és O(n).
  - Si el valor no hi és, farem n comparacions.
  - Si el valor hi és, farem una mitja de n/2 comparacions.
- Si la llista està ordenada farem cerca binaria:
  - La cerca binaria té una complexitat O(log n)
  - La complexitat és la mateixa tant si hi és com si no.
- Com podem millorar-ho?

- Suposem que tenim una “funció màgica” que, donat un valor a cercar, ens digués a quina posició de la llista mirar, i:
  - Si hi és, és que estava a la llista.
  - Si no hi és, és que no hi era.

- Aquesta funció s’anomena **funció hash**.

Funcions Hash i Cerca

Imaginem que volem assignar un nom "curt" (el més curt possible) a cada una de les possibles 2^{32} adreces IP.

Inevitablement, en general, moltes adreces tindrien el mateix nom! Però pot tenir sentit si treballem amb les adreces d'una sola empresa.

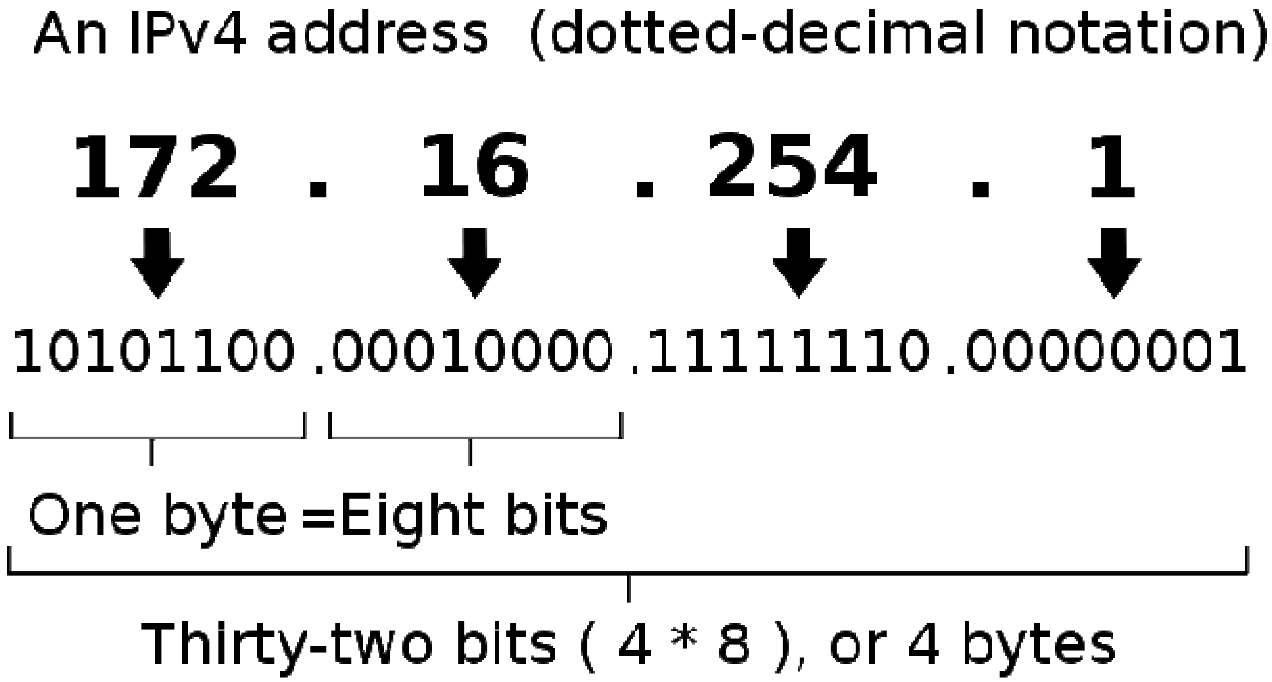
Imaginem que en tenim prou amb 250 noms.

Aquest és el nombre esperat de dades.

En el cas d'assignar el mateix nom a dues adreces, o **col·lisió**, associem a aquell nom una llista d'IPs amb aquell nom.

Aquest esquema té sentit per fer cerques molt ràpides, atès que les llistes de col·lisió seran petites.

The designers of TCP/IP defined an IP address as a 32-bit number and this system, known as Internet Protocol Version 4 (IPv4), is still in use today. However, due to the enormous growth of the Internet and the predicted depletion of available addresses, a new addressing system (IPv6), using 128 bits for the address, was developed in 1995, standardized by RFC 2460 in 1998, and is in world-wide production deployment.



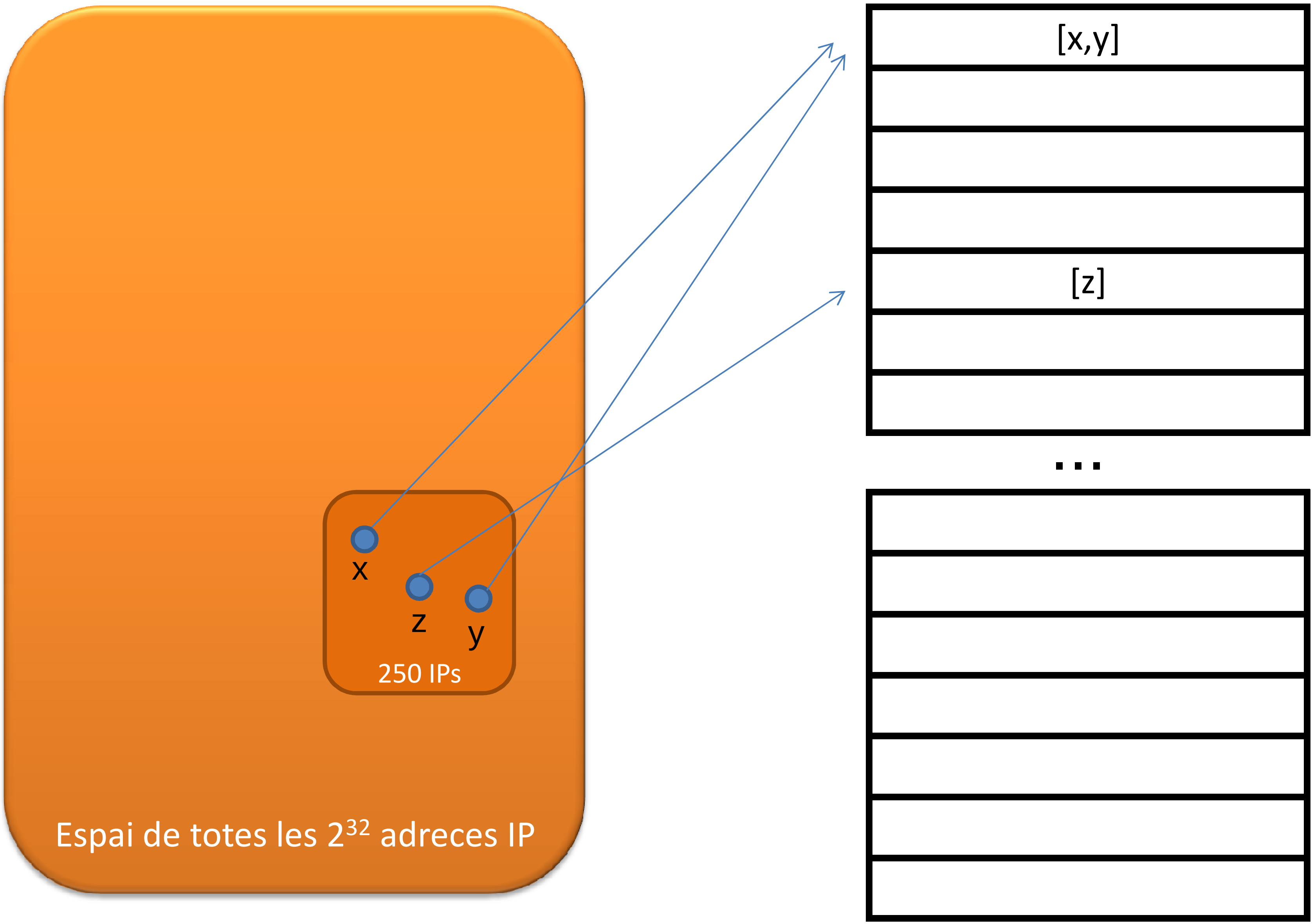
Funcions Hash i Cerca

Com generem (amb una **funció**) les posicions per minimitzar la probabilitat de col·lisió?

Aquest és el rol de les **funcions *hash***: una funció *h* que fa assigna adreces IP a posicions d'una taula de longitud 250.

El nom assignat a una adreça *x* és *h(x)*, i llavors *x* s'emmagatzema a la posició *h(x)* de la taula.

Cada posició és una llista si volem manegar les col·lisions!



[x,y]
[z]
...
x
z
y
250 IPs
Espai de totes les 2^32 adreces IP

Funcions *Hash* i Cerca

Les funcions *hash* han de ser:

- **Aleatòries**, en el sentit que ha de distribuir les dades per tot arreu.
- **Consistents**, en el sentit d'assignar a cada ítem sempre el mateix nom.

Imaginem que usem una funció que assigna l'adreça al nombre corresponent als últims 8 bits:

$$h(128.32.168.80)=80$$

És una bona funció *hash*?

Funcions *Hash* i Cerca

No!

Les posicions corresponents als nombres baixos estaran molt plens, atès que en la pràctica aquests nombres s’assignen successivament.

I si posem el primer segment?

Tampoc!

Imaginem que la majoria d’adreces són d’Asia.

Si les adreces **vinguessin de forma uniformement distribuïda del conjunt total** no hi hauria problema. Però això no passa mai.

Funcions *Hash* i Cerca

Atès que fem una correspondència entre 2^{32} adreces i una llista de 250 noms hi ha d'haver una col·lecció de $2^{32}/250 = 16.000.000$ d'adreces que col·lisionaran.

Però podem seguir el següent esquema per minimitzar les col·lisions: escollim de forma aleatòria una funció per alguna classe de funcions *hash* (per una funció *hash* haurem de poder demostrar que, sigui quin sigui el conjunt d'adreces, la majoria de funcions escollides generaran poques col·lisions!)

Funcions *Hash* i Cerca

Anem a definir una classe de funcions que puguem escollir aleatòriament:

- **Suposem que creem una llista de *n*=257 noms (enlloc de 256). 257 és un nombre primer.....**
- **Representem una adreça com una quàdruple d'enters mòdul *n*: $x = (x_1,x_2,x_3,x_4)$**

Definim la classe *h: IP -> n* de la següent manera:

- **Defineix 4 nombres qualsevol mòdul *n*=257. Per exemple: 87, 23, 125, 4.**
- **Transforma l'adreça (x_1,x_2,x_3,x_4) a**

$$h(x_1,x_2,x_3,x_4)= (87x_1+23x_2+125x_3+4x_4) \bmod 257$$

Funcions *Hash* i Cerca

Dit d'una altra manera:

Per qualsevol conjunt de coeficients

$$a_1,\ldots,a_4 \in \{0,1,\ldots,n-1\}$$

escriurem

$$a = (a_1,\ldots,a_4)$$

i definirem la següent funció *hash*:

$$h_a(x_1,\ldots,x_4) = \sum_{i=1}^{4} a_i \cdot x_i \bmod n$$

Es pot demostrar que:

Donada qualsevol parella d'adreces IP $x = (x_1,x_2,x_3,x_4)$ i $y = (y_1,y_2,y_3,y_4)$, si els coeficients (a_1,a_2,a_3,a_4) s'escullen aleatòriament de $\{0,1,\ldots,n-1\}$, llavors:

$$\Pr\{h_a(x_1,\ldots,x_4) = h_a(y_1,\ldots,y_4)\} = \frac{1}{n}$$

Sigui quin sigui el conjunt d'adreces, la majoria de funcions escollides generaran poques col·lisions!

És a dir, la mateixa *Pr* que si les adreces haguessin estat escollides de forma aleatòria.

Proof. Since $x = (x_1, \ldots, x_4)$ and $y = (y_1, \ldots, y_4)$ are distinct, these quadruples must differ in some component; without loss of generality let us assume that $x_4 \neq y_4$. We wish to compute the probability $\Pr[h_a(x_1, \ldots, x_4) = h_a(y_1, \ldots, y_4)]$, that is, the probability that $\sum_{i=1}^{4} a_i \cdot x_i \equiv \sum_{i=1}^{4} a_i \cdot y_i \bmod n$. This last equation can be rewritten as

$$\sum_{i=1}^{3} a_i \cdot (x_i - y_i) \equiv a_4 \cdot (y_4 - x_4) \bmod n \tag{1}$$

Suppose that we draw a random hash function h_a by picking $a = (a_1, a_2, a_3, a_4)$ at random. We start by drawing a_1, a_2, and a_3, and then we pause and think: What is the probability that the last drawn number a_4 is such that equation (1) holds? So far the left-hand side of equation (1) evaluates to some number, call it c. And since n is prime and $x_4 \neq y_4$, $(y_4 - x_4)$ has a unique inverse modulo n. Thus for equation (1) to hold, the last number a_4 must be precisely $c \cdot (y_4 - x_4)^{-1} \bmod n$, out of its n possible values. The probability of this happening is $1/n$, and the proof is complete. ∎

Funcions *Hash* i Cerca

Quin és el **temps de cerca** d'una adreça?

El càlcul de la funció *hash* + la cerca en la llista assignada al mateix nom.

Però només hi ha 250 adreces i la probabilitat de col·lisió és 1/257. Per tant, el nombre esperat d'ítems emmagatzemats al mateix nom és 250/257 (o sigui, pel nostre problema, menys de dos).

Funcions *Hash* i Cerca

1.3.2 Hash Functions

The reader has probably heard of hash tables, and perhaps used them in Java classes or similar packages. The hash functions that make hash tables feasible are also essential components in a number of data-mining algorithms, where the hash table takes an unfamiliar form. We shall review the basics here.

First, a hash function h takes a *hash-key* value as an argument and produces a *bucket number* as a result. The bucket number is an integer, normally in the range 0 to $B - 1$, where B is the number of buckets. Hash keys can be of any type. There is an intuitive property of hash functions that they "randomize" hash-keys. To be precise, if hash-keys are drawn randomly from a reasonable population of possible hash-keys, then h will send approximately equal numbers of hash-keys to each of the B buckets. It would be impossible to do so if, for example, the population of possible hash-keys were smaller than B. Such a population would not be "reasonable." However, there can be more subtle reasons why a hash function fails to achieve an approximately uniform distribution into buckets.

Example 1.4: Suppose hash keys are positive integers. A common and simple hash function is to pick $h(x) = x \mod B$, that is, the remainder when x is divided by B. That choice works fine if our population of hash-keys is all positive integers. $1/B$th of the integers will be assigned to each of the buckets. However, suppose our population is the even integers, and $B = 10$. Then only buckets 0, 2, 4, 6, and 8 can be the value of $h(x)$, and the hash function is distinctly nonrandom in its behavior. On the other hand, if we picked $B = 11$, then we would find that 1/11th of the even integers get sent to each of the 11 buckets, so the hash function would work very well. □

The generalization of Example 1.4 is that when hash-keys are integers, choosing B so it has any common factor with all (or even most of) the possible hash keys will result in nonrandom distribution into buckets. Thus, it is normally preferred that we choose B to be a prime. That choice reduces the chance of nonrandom behavior, although we still have to consider the possibility that all hash-keys have B as a factor. Of course there are many other types of hash functions not based on modular arithmetic. We shall not try to summarize the options here, but some sources of information will be mentioned in the bibliographic notes.

Funcions *Hash* i Cerca

Recapitulem el que hem fet:

- Com que no teníem control sobre el conjunt de dades que ens arribava, hem escollit una funció *h* de forma uniformement aleatòria d'entre una família *H*. En el nostre cas: $H = \{h_a : a \in \{0, \ldots, n-1\}^4\}$
  - Per escollir-la, hem escollit aleatòriament 4 nombres mòdul *n*.
- Hem dit que es pot demostrar que per dos ítems qualsevol *x* i *y*, aquests aniran al mateix lloc **amb una probabilitat** $1/n$, on *n* és el nombre de llocs.

Una família de funcions *hash* amb aquesta propietat es diu **universal**.

La seva aplicació a d'altres problemes és simple:

• Triem una taula de mida *n* tal que *n* sigui un nombre primer una mica més gran que el nombre d'elements de la taula (o millor encara, el doble).

• Assumim que el domini dels ítems és $N=n^k$ (encara que ho sobreestimem).

• Llavors cada ítem es representa amb una *k*-tupla d'enters mòdul *n*, i $H=\{h_a : a\in\{0,\ldots,n-1\}^k\}$ és una família universal de funcions *hash*.

Famílies

La natura de les famílies depèn de les dades!

Trivial hash function: If the datum to be hashed is small enough, one can use the datum itself (**reinterpreted as an integer in binary notation**) as the hashed value. The cost of computing this "trivial" (identity) hash function is effectively zero.

The meaning of "small enough" depends on how much memory is available for the hash table. A typical PC might have 4 gigabytes of available memory, meaning that hash values of up to 32 bits could be accommodated.

However, there are many applications that can get by with much less. For example, when **mapping character strings between upper and lower case**, one can use the binary encoding of each character, interpreted as an integer, to index a table that gives the alternative form of that character ("A" for "a", "8" for "8", etc.). If each character is stored in 8 bits (as in ASCII or ISO Latin 1), the table has only $2^8 = 256$ entries.

The same technique can be used to map two-letter country codes like "us" or "za" to country names (26^2=676 table entries), 5-digit zip codes like 13083 to city names (100000 entries), etc. Invalid data values (such as the country code "xx" or the zip code 00000) may be left undefined in the table, or mapped to some appropriate "null" value.

Families

La natura de les famílies depèn de les dades!

Hashing uniformly distributed data:

If the inputs are bounded-length strings (such as telephone numbers, car license plates, invoice numbers, etc.), and each input
may independently occur with uniform probability, then a hash function need only map roughly the same number of inputs to each hash value.

For instance, suppose that each input is an integer z in the range 0 to $N-1$, and the output must be an integer h in the range 0 to $n-1$, where N is much larger than n. Then the hash function could be $h = z$ **mod** n (the remainder of z divided by n), or $h = (z \times n) \div N$ (the value z scaled down by n/N and truncated to an integer), or many other formulas.

Families

La natura de les famílies depèn de les dades!

Hashing data with other distributions

These simple formulas will not do if the input values are not equally likely, or are not independent. For instance, most patrons of a supermarket will live in the same geographic area, so their telephone numbers are likely to begin with the same 3 to 4 digits.

In that case, if n is 10000 or so, the division formula $(z \times n) \div N$, which depends mainly on the leading digits, will generate a lot of collisions; whereas the remainder formula $z \ \mathbf{mod}\ n$, which is quite sensitive to the trailing digits, may still yield a fairly even distribution of hash values.

Families

La natura de les famílies depèn de les dades!

Hashing variable-length data

When the data values are long (or variable-length) character strings—such as personal names, web page addresses, or mail messages—their distribution is usually very uneven, with complicated dependencies.

For such data, it is prudent to use a hash function that depends on all characters of the string—and depends on each character in a different way.

In general, the scheme for hashing such data is to break the input into a sequence of small units (bits, bytes, words, etc.) and combine all the units $b[1]$, $b[2]$, ..., $b[m]$ sequentially.

```
def DEKHash(key):
    hash = len(key);
    for i in range(len(key)):
      hash = ((hash << 5) ^ (hash >> 27)) ^ ord(key[i])
    return hash
```

Exemple:

```
def DEKHash(key):
    hash = len(key);
    for i in range(len(key)):
      hash = ((hash << 5) ^ (hash >> 27)) ^ ord(key[i])
    return hash
>>> DEKHash('algorismica')
291014770516511749L
```

a = 0011 1100
b = 0000 1101

^	Binary XOR Operator: copies the bit if it is set in one operand but not both. (a ^ b) will give 49 which is 0011 0001
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. a << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. a >> 2 will give 15 which is 0000 1111

Exemple:

`m` representa el nombre esperat d'ítems a guardar.

`C` és un nombre més gran que la qualsevol `ord(c)`

```
def StringHash(a, m=257, C=1024):
    hash=0
    for i in range(len(a)):
        hash = (hash * C + ord(a[i])) % m
    return hash

>>> StringHash('hola')
182
>>> StringHash('adeu')
245
>>> StringHash('adeu hol adeu')
208
>>> StringHash('a')
97
```

Aquí ho apliquem a caràcters, però ho podríem fer amb bigrames, etc.

Exemple:

```
def StringHash(a, m=257, C=1024):
    hash=0
    for i in range(len(a)):
        hash = (hash * C + ord(a[i])) % m
    return hash
```

```
>>> diccionari
['the', 'and', 'to', 'of', 'a', 'I', 'in', 'was', 'he', 'that', 'it', 'his',
'her', 'you', 'as', 'had', 'with', 'for', 'she', 'not', 'at', 'but', 'be', 'my',
'on', 'have', 'him', 'is', 'said', 'me', 'which', 'by', 'so', 'this', 'all',
'from', 'they', 'no', 'were', 'if', 'would', 'or', 'when', 'what', 'there',
'been', 'one', 'could', 'very', 'an', 'who', 'them', 'Mr', 'we', 'now', 'more',
'out', 'do', 'are', 'up', 'their', 'your', 'will', 'little', 'than', 'then',
'some', 'into', 'any', 'well', 'much', 'about', 'time', 'know', 'should', 'man',
'did', 'like', 'upon', 'such', 'never', 'only', 'good', 'how', 'before',
'other', 'see', 'must', 'am', 'own', 'come', 'down', 'say', 'after', 'think',
'made', 'might', 'being', 'Mrs', 'again']
>>> taula=[]
>>>for i in diccionari:
     taula.append(StringHash(i))
>>> taula
[256, 184, 161, 172, 97, 73, 204, 89, 199, 136, 210, 74, 89, 67, 241, 91, 129,
17, 240, 147, 242, 188, 223, 199, 180, 179, 68, 209, 40, 179, 10, 243, 165, 103,
200, 101, 125, 185, 70, 196, 228, 184, 179, 201, 143, 190, 152, 248, 154, 236,
57, 113, 63, 139, 150, 99, 139, 225, 169, 158, 192, 103, 165, 82, 130, 114, 249,
84, 205, 101, 1, 195, 89, 241, 189, 181, 252, 95, 138, 131, 164, 256, 237, 54,
67, 7, 252, 206, 235, 125, 245, 150, 31, 81, 229, 188, 173, 178, 120, 207]
```

Exemple:

```
def StringHash(a, m=5749, C=1024):
    hash=0
    for i in range(len(a)):
        hash = (hash * C + ord(a[i])) % m
    return hash
```

```
>>> taula
[589, 4073, 3915, 4535, 97, 73, 4148, 209, 3115, 1260,
4154, 3276, 4928, 1816, 1710, 818, 3248, 4903, 4080,
5722, 1711, 2017, 2720, 2506, 4543, 5313, 3270, 4153,
4034, 2486, 827, 2740, 2891, 3702, 2033, 5606, 5361,
3520, 3663, 4140, 4550, 4547, 2883, 4542, 3800, 1880,
1192, 3283, 2589, 1705, 1624, 5349, 4225, 1228, 5725,
3805, 2626, 4778, 2421, 4940, 346, 2771, 809, 824,
1254, 5350, 5249, 4978, 4094, 3275, 1996, 3590, 4293,
2054, 2931, 620, 5727, 4991, 254, 2811, 1654, 3360,
5666, 3675, 1738, 2900, 1008, 1145, 1704, 4668, 4992,
4837, 2681, 2870, 2993, 3849, 4778, 2591, 3267, 5397]
```

SimHash i cerca per semblança.

Imaginem que tenim un conjunt d'adreces. Com podem decidir que són repetides?

És evident que no són la mateixa

Cas 1:

Burra Hotel, 5 Market Sq, Burra, SA, 5417

Camping Country Superstore, 401 Pacific Hwy, Belmont North, NSW, 2280

Cas 2:

Segurament són la mateixa

One Stop Bakery, 1304 High St Rd, Wantirna, VIC, 3152

One Stop Bakery, 1304 High Street Rd, Wantirna South, VIC, 3152

Segurament són la mateixa

Cas 3:

Park Beach Interiors, Showroom Park Beach Plaza Pacific Hwy, Coffs Harbour, NSW, 2450

Park Beach Interiors, Showroom Park Beach Plaza Pacific Highway, Coffs Harbour, NSW, 2450

Park Beach Interiors, Park Beach Plaza Pacific Hwy, Coffs Harbour, NSW, 2450

Park Beach Interiors, 26 Park Beach Plaza, Pacific Hwy, Coffs Harbour, NSW, 2450

SimHash i cerca per semblança.

Imaginem que tenim un conjunt d'adreces. Com podem decidir que són repetides?

Podrien ser la mateixa, o no!

Cas 4:
Weaver Interiors, 955 Pacific Hwy, Pymble, NSW, 2073
Weaver Interiors, 997 Pacific Hwy, Pymble, NSW, 2073

Podrien ser la mateixa, o no!

Cas 5:
Gibbon Hamor Commercial Interiors, 233 Johnston St, Annandale, NSW, 2038
Gibbon Hamor Development Planners, 233 Johnston St, Annandale, NSW, 2038

Necessitem una manera de "comparar" aquestes adreces.

SimHash i cerca per semblança.

Les tècniques de *shingling* són una forma de generar un conjunt que pot ser usat com a representant del text.

Per exemple, podem usar **bigrames** de paraules:

Els bigrames que representen `"the cat sat on the“` són...

```
(the,cat)
(cat,sat)
(sat,on)
(on,the)
```

SimHash i cerca per semblança.

Llavors podem aplicar **l'índex de Jaccard** **(el quocient entre la cardinalitat de la intersecció de dos conjunt i la cardinalitat de la unió)** com a mesura de semblança entre adreces!

Si J(A,B) és l'índex de Jaccard entre A i B, i tenim A = {1,2,3}, B = {2,3,4}, C = {4,5,6}, llavors,

J(A,B) = 2/4 = 0.5,
J(A,C) = 0/6 = 0,
J(B,C) = 1/5 = 0.2

I els conjunts més semblants són A i B I els menys semblants són A i C.

SimHash i cerca per semblança.

`Burra Hotel, 5 Market Sq, Burra, SA, 5417`

es pot representar (amb bigrames de lletres) com:

```
A={" 5", " B", " H", " M", " S", ", ", "17", "41", "5 ",
"54", "A,", "Bu", "Ho", "Ma", "SA", "Sq", "a ", "a,", "ar",
"el", "et", "ke", "l,", "ot", "q,", "ra", "rk", "rr", "t ",
"te", "ur"}
```

`Camping Country Superstore, 401 Pacific Hwy, Belmont North, NSW, 2280`

es pot representar (amb bigrames de lletres) com:

```
B={" 2", " 4", " B", " C", " H", " N", " P", " S", ", ",
"01", "1 ", "22", "28", "40", "80", "Be", "Ca", "Co", "Hw",
"NS", "No", "Pa", "SW", "Su", "W,", "ac", "am", "c ", "ci",
"e,", "el", "er", "fi", "g ", "h,", "ic", "if", "in", "lm",
"mo", "mp", "ng", "nt", "on", "or", "ou", "pe", "pi", "re",
"rs", "rt", "ry", "st", "t ", "th", "to", "tr", "un", "up",
"wy", "y ", "y,"}
```

$$J(A,B) = 6/87 = 0.068$$

SimHash i cerca per semblança.

Cas 2

```
A= One Stop Bakery, 1304 High St Rd, Wantirna, VIC, 3152
B= One Stop Bakery, 1304 High Street Rd, Wantirna South, VIC,
3152
```

J(A,B)=46/57 = 0.807

Cas 3

```
A= Park Beach Interiors, Showroom Park Beach Plaza Pacific Hwy,
Coffs Harbour, NSW, 2450
B= Park Beach Interiors, Showroom Park Beach Plaza Pacific
Highway, Coffs Harbour, NSW, 2450
C= Park Beach Interiors, Park Beach Plaza Pacific Hwy, Coffs
Harbour, NSW, 2450
D= Park Beach Interiors, 26 Park Beach Plaza, Pacific Hwy, Coffs
Harbour, NSW, 2450
```

J(A,B)=0.888, J(A,C)=0.861, J(A,D)=0.808, J(B,C)=0.760,
J(B,D)=0.716, J(C,D)=0.932

SimHash i cerca per semblança.

Cas 4

A = `Weaver Interiors, 955 Pacific Hwy, Pymble, NSW, 2073`
B = `Weaver Interiors, 997 Pacific Hwy, Pymble, NSW, 2073`

J(A,B)=43/49 = 0.877

Cas 5

A= `Gibbon Hamor Commercial Interiors, 233 Johnston St, Annandale, NSW, 2038 and`
B= `Gibbon Hamor Development Planners, 233 Johnston St, Annandale, NSW, 2038`

J(A,B)= 49/76 = 0.644

SimHash i cerca per semblança.

Si volem buscar el parell d'adreces més semblants tenim un problema: hem de comparar cada element amb tots els elements i això és O(n^2).

50 adreces són	1,225 comparacions.
100 adreces són	4,950 comparacions
250 adreces són	31,125 comparacions
500 adreces són	124,750 comparacions
750 adreces són	280,875 comparacions
2000 adreces són	1,999,000 comparacions
1,000,000 adreces són	499,999,500,000 comparacions.

Podem calcular un valor Sim(A,B) que "aproximi" J(A,B) amb menys complexitat que $O(n^2)$?

SimHash i cerca per semblança.

Per fer-ho necessitem una funció de *hash* que assigni codis semblants (en el sentit de la **distància de Hamming**) a ítems semblants.

Hash clàssic vs **SimHash**:

```
>>> p1 = 'the cat sat on the mat'
>>> p2 = 'the cat sat on a mat'
>>> p3 = 'we all scream for ice cream'
>>> hash(p1)
415542861
>>> hash(p2)
668720516
>>> hash(p3)
767429688
>>> simhash(p1)
851459198
00110010110000000011110001111110
>>> simhash(p2)
847263864
00110010100000000011100001111000
>>> simhash(p3)
984968088
00111010101101010110101110011000
```

hamming(p1,p2)=4
hamming(p1,p3)=16
hamming(p2,p3)=12

SimHash i cerca per semblança.

Distància de Hamming:

In information theory, the **Hamming distance** between two strings of equal length is the number of positions at which the corresponding symbols are different. Put another way, it measures the minimum number of *substitutions* required to change one string into the other, or the number of *errors* that transformed one string into the other.

The Hamming distance between:
"**toned**" and "**roses**" is 3.
1011101 and **1001001** is 2.
2173896 and **2233796** is 3.

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables.

```
def hamming_distance(s1, s2):
        return sum(ch1 != ch2 for ch1, ch2 in zip(s1, s2))

>>> zip('hola','hole')
[('h', 'h'), ('o', 'o'), ('l', 'l'), ('a', 'e')]
>>> hamming_distance('hola','hole')
1
```

SimHash i cerca per semblança.

L'algorisme SimHash genera un codi de la manera següent:

1. Escollim una mida de la taula, per exemple 32 bits.
2. Sigui **`V=[0]*32`.**
3. Representem el text amb els bigrames:

```
>>> s = set( )
>>> text = 'the quick brown fox jumps over the lazy dog.'
>>> for i in range( len( text ) ): s.add( text[ i: i + 2 ] )
>>> print s
set(['ck', ' b', ' f', 've', ' d', ' j', 'y ', ' o', ' l', ' q', 'zy', ' t', 'ic', 'az', 'er', 'ps', 'he', 'k
', 'la', '.', 'th', 'ju', 'r ', 'ro', 'do', 'wn', 'x ', 'e ', 'br', 'g.', 'fo', 's ', 'qu', 'n ', 'um',
'og', 'ui', 'mp', 'ox', 'ow', 'ov'])
```

```
'the cat sat on the mat' =>
{"th", "he", "e ", " c", "ca", "at", "t ", " s", "sa", " o",
"on", "n ", " t", " m", "ma"}
```

4. Fem el hash de cada característica amb una funció de 32 bits.

```
Hash('th') = -502157718
Hash('he') = -369049682 ...
```

5. Per cada hash, si el bit_i del hash és 1 llavors sumem 1 a V[i]; si el bit_i del hash és 0 llavors restem 1 de V[i].

6. El bit_i de simhash és 1 si V[i] > 0 i 0 en els altres casos.

SimHash i cerca per semblança.

Simhash és útil perquè **si la distància de Hamming entre els simhash de dues frases és petit, llavors el coeficient de Jaccard és alt**. Però com ho usem pel nostre objectiu?

Observació:

En el cas que 2 nombres tenen una distància de Hamming petita i la seva diferència està en els bits menys significatius, llavors això vol dir que són dos ítems que estan propers en una llista ordenada.

Sense ordenar

1	37586	1001001011010010	
2	50086	1100001110100110	7
3	2648	0000101001011000	11
4	934	0000001110100110	9
5	40957	1001111111111101	9
6	2650	0000101001011010	9
7	64475	1111101111011011	7
8	40955	1001111111111011	4

Distància de Hamming al nombre anterior.

Ordenats

4	934	0000001110100110	
3	2648	0000101001011000	9
6	2650	0000101001011010	1
1	37586	1001001011010010	5
8	40955	1001111111111011	6
5	40957	1001111111111101	2
2	50086	1100001110100110	9
7	64475	1111101111011011	9

SimHash i cerca per semblança.

Per tant, només cal comprovar parelles adjacents de la llista per trobar la parella més semblant!

Això redueix la complexitat de *n*(n-1)/2* comparacions amb $O(n^2)$ a **n càlculs de la funció *hash* amb *O(n)* + una ordenació *O(n logn)* + *n* distàncies amb *O(n)*** : la complexitat total és ***O(n logn).***

4	934	0000001110100110	
3	2648	0000101001011000	9
6	2650	0000101001011010	1
1	37586	1001001011010010	5
8	40955	1001111111111011	6
5	40957	1001111111111101	2
2	50086	1100001110100110	9
7	64475	1111101111011011	9

SimHash i cerca per semblança.

Però hi ha una parella semblant que ha deixat apartats: el 4 i el 2, que tenen distància = 2...

L'ordenació només ha ajuntat els que tenien bits poc significatius semblants...

Però podem aprofitar una propietat de la distància de Hamming: **una permutació dels bits dels nombres preserva la distància!**

```
Ham(100010,110010) = Ham(001010,001011)
```

Si permutem mitjançant la rotació dels bits tenim la mateixa distància però els bits més significatius passen a ser els menys significatius.

SimHash i cerca per semblança.

Però hi ha una parella semblant que ha deixat apartats: el 4 i el 2, que tenen distància = 2...

Rotem els bits a l'esquerra dues vegades i després ordenem:

4	3736	0000111010011000	
3	10592	0010100101100000	9
6	10600	0010100101101000	1
1	19274	0100101101001010	5
8	32750	0111111111101110	6
5	32758	0111111111110110	2
2	3739	0000111010011011	9
7	61295	1110111101101111	9

4	3736	0000111010011000	
2	3739	0000111010011011	2
3	10592	0010100101100000	11
6	10600	0010100101101000	1
1	19274	0100101101001010	5
8	32750	0111111111101110	6
5	32758	0111111111110110	2
7	61295	1110111101101111	6

SimHash i cerca per semblança.

Per tant, l'algorisme ha de desplaçar els bits B vegades (on *B* és la longitud en bits) i fer:

1. Rotar els bits
2. Ordenar
3. Calcular la distància de Hamming entre files adjacents.

Com que segurament $B \ll n$ el cost de l'algorisme és més semblant a $O(n \log n)$ que a $O(n^2)$.