Formation ISN niveau 2

Les dictionnaires

Michel Van Caneghem
michel.vancaneghem@univ-amu.fr

Janvier 2013

Aix*Marseille université

Les dictionnaires

C'est une structure de données importante dans Python. Nous allons examiner :

- Comment chercher un mot dans un dictionnaire
- Comment on peut implanter une telle structure de données
- Pourquoi cela marche !!
- Différentes utilisation des dictionnaires
- Réalisation d'un correcteur orthographique (le prochain TP)

Ceci est un cours aura beaucoup de digression. Le matériel complet est disponible ici :

`http://michel.vancaneghem.perso.luminy.univ-amu.fr/cours/isn/dictionnaire.zip`

Un dictionnaire

La liste des mots du français que je vous propose (dico2012.txt) est un dictionnaire de 424579 mots (formes fléchies comprises et quelques noms propres) – ce lexique est extrait du site : http ://www.dicollecte.org/. (Cette liste est codée en UTF8).

Pour savoir si un mot est bien orthographié on cherche s'il appartient oui ou non à notre liste de mots. Par exemple :

- « omnibulé » n'appartient pas à la liste des mots.
- « obnubilé » appartient à la liste des mots.

C'est cette opération que nous allons analyser

Lecture du dictionnaire

Voici comment on peut lire cette liste en Python

Lecture du dictionnaire

```
listeMots = [ligne.strip() for ligne in open("dico2012.txt", 'r')]
```

Il faut insister sur cette manière d'écrire : « *list comprehensions* ». Qui se présente en général sous la forme : `[expression for element in liste]` ou `expression` est une fonction de `element`.

un exemple

```
# Affiche les carrés des éléments
liste = [1, 2, 3, 4, 5, 6, 7]
print([x ** 2 for x in liste])
```

Les listes en Python

C'est la structure de données la plus utilisée en Python. En fait il faut bien comprendre qu'une « list » en Python est *un tableau de longueur variable et non une liste traditionnelle avec des pointeurs*. Ceci a pour conséquence :

- Que l'accès a un élément se fait en temps constant $O(1)$ et se note de manière traditionnelle `a[i]`.
- Que l'insertion ou la suppression d'un élément se fait en temps $O(n)$ si n est la longueur de la liste : pour ajouter un élément on décale tous les éléments suivants vers la droite.
- Que l'ajout ou la suppression d'un élément en bout de liste se fait en temps constant $O(1)$.

On préalloue de la mémoire pour une liste et si il manque de place on double la taille de l'allocation mémoire.

Appartenance au dictionnaire

Cela est très simple :

in

```
print('omnibulé' in listeMots) #False -- 6.8 ms
print('obnubilé' in listeMots) #True  -- 4.2 ms
```

En fait on parcourt séquentiellement la liste des mots (toute la liste dans le premier cas) en comparant notre mot à tous les mots de la liste : $O(n)$ si n est la longueur de la liste. Il faut donc faire 424579 comparaisons.

Cela peut paraître rapide, mais si on veut analyser un roman (60 000 mots), cela va mettre 420 secondes !!

Comment mesurer le temps

timeit

```
from timeit import timeit
timeit(stmt="'omnibulé' in liste",
       setup="liste = %r" % listeMots,
       number=10000)
```

- `timeit` est une librairie Python ;
- `stmt` représente l'instruction dont on va mesurer le temps d'exécution ;
- `setup` représente les instructions qui vont être exécutées initialement (une fois) – ne cherchez pas à comprendre la syntaxe : il faut savoir que `timeit` recrée un environnement d'exécution et qu'il ne connaît pas les variables du programmes qui contient l'appel de `timeit` ;
- `number` est le nombre de fois que l'énoncé est exécuté (par défaut 1000000).

Un vrai dictionnaire

En fait dans un vrai dictionnaire les mots sont triés.

tri

```
listeMots.sort()
```

On va utiliser la même méthode que pour une recherche manuelle dans un dictionnaire : on recherche le mot en le comparant au mot du milieu de la liste et si il est plus petit on recommence avec la partie gauche et sinon avec la partie droite. Quand il n'y a plus qu'un mot : ou bien c'est le bon, ou bien le mot cherché n'appartient pas au dictionnaire.

Il faut $O(\log_2 n)$ comparaisons. Dans notre exemple : $2^{18} = 262144$ et $2^{19} = 524288$. Il nous faut donc 19 comparaisons pour trouver le mot. Soit 20000 fois plus rapide !

La recherche dichotomique

binarySearch

```
def binarySearch(seq, t) :
    left = 0 ; right = len(seq) - 1
    while True :
        if right < left :
            return -1
        m = (left + right) // 2
        if seq[m] < t :
            left = m + 1
        elif seq[m] > t :
            right = m - 1
        else :
            return m
```

Le temps d'exécution

binarySearch

```
print(binarySearch(listeMots,'omnibulé')) # -1     -- 5.6µs
print(binarySearch(listeMots,'obnubilé')) # 265620 -- 5.6µs
```

Seulement 1000 fois plus rapide ! En fait cela vient du fait que Python est un langage interprété. Si on utilise la fonction : `bisect_left` d'une librairie Python (qui fait presque la même chose) alors on trouve :

bisect_left

```
print(bisect_left(listeMots,'omnibulé')) # 267653 -- 0.65µs
print(bisect_left(listeMots,'obnubilé')) # 265620 -- 0.74µs
```

Ce qui est plus raisonnable

Le vrai dictionnaire en Python

dict

```
dico = dict.fromkeys(listeMots)

print('omnibulé' in dico) #False -- 40ns
print('obnubilé' in dico) #True  -- 48ns
```

Soit 10 fois plus rapide que la recherche binaire. L'analyse de notre roman devrait se faire en 3 ms !!

Le reste de cet exposé va montrer comment et pourquoi cela marche

Les dictionnaires Python

un dictionnaire est un ensemble de couples cle :valeur, comme un tableau est un ensemble de couples indice :valeur. On accède a un élément par sa clé, de manière efficace($O(1)$), mais quand même moins efficace qu'un tableau ordinaire. L'ordre des éléments dans un dictionnaire n'est pas significatif.

dictionnaire

```
>>> animaux = {'veau' :13.50, 'vache' :12, 'poulet' :6.5}
>>> animaux['poulet']
6.5
>>> animaux['lapin'] = 7
>>> animaux
{'poulet' : 6.5, 'vache' : 12, 'veau' : 13.5, 'lapin' : 7}
>>> 'lapin' in animaux
True
>>> animaux.keys()
dict_keys(['poulet', 'vache', 'veau', 'lapin'])
>>> animaux.values()
dict_values([6.5, 12, 13.5, 7])
```

Une fonction de hachage

Un dictionnaire est une liste. Pour savoir quelle case remplir, on passe par une fonction des objets (ici des chaînes) vers les entiers : la fonction de « hashcode » (hachage).

dictionnaire

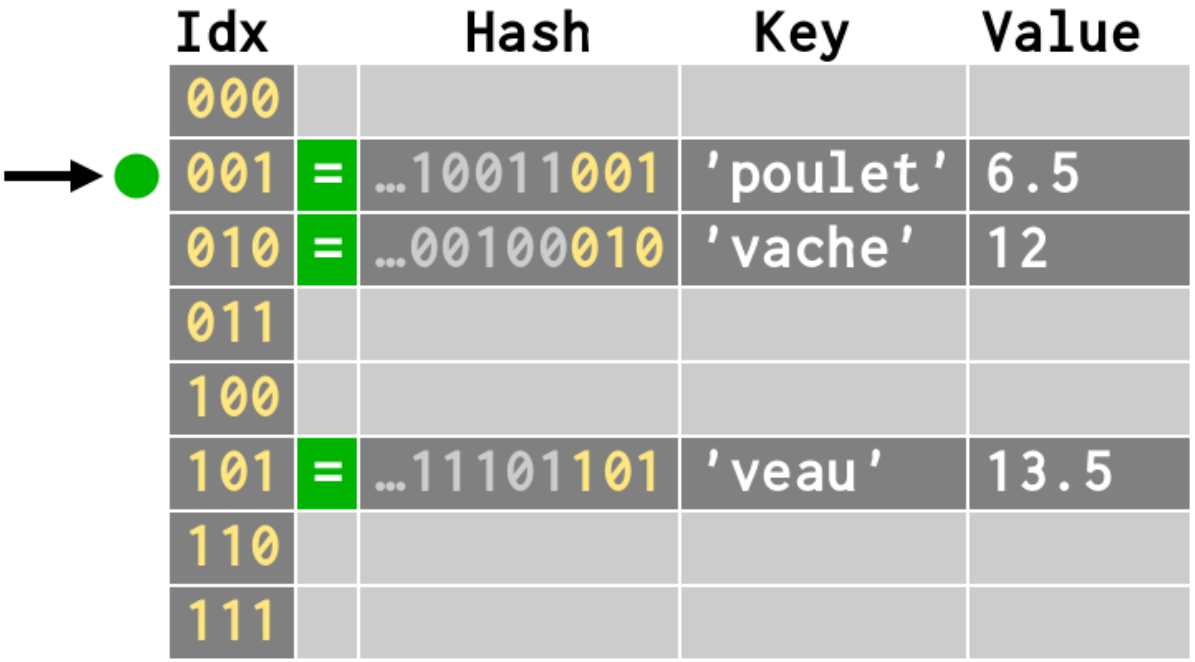
```
hash('veau')          -7823613590878614035
hash('vache')         -7965211691920464862
hash('poulet')        -4033560038549015655
hash('cochon')         8775908623350139890
hash('veau') % 8       5
hash('vache') % 8      2
hash('poulet') % 8     1
hash('cochon') % 8     2
```

Cette fonction doit répartir les valeurs de manière raisonnable

La table de hashcode ou le dictionnaire (1)

Regardons ce qui se passe après l'instruction suivante ;

```
animaux = {'veau' :13.50, 'vache' :12, 'poulet' :6.5}
```

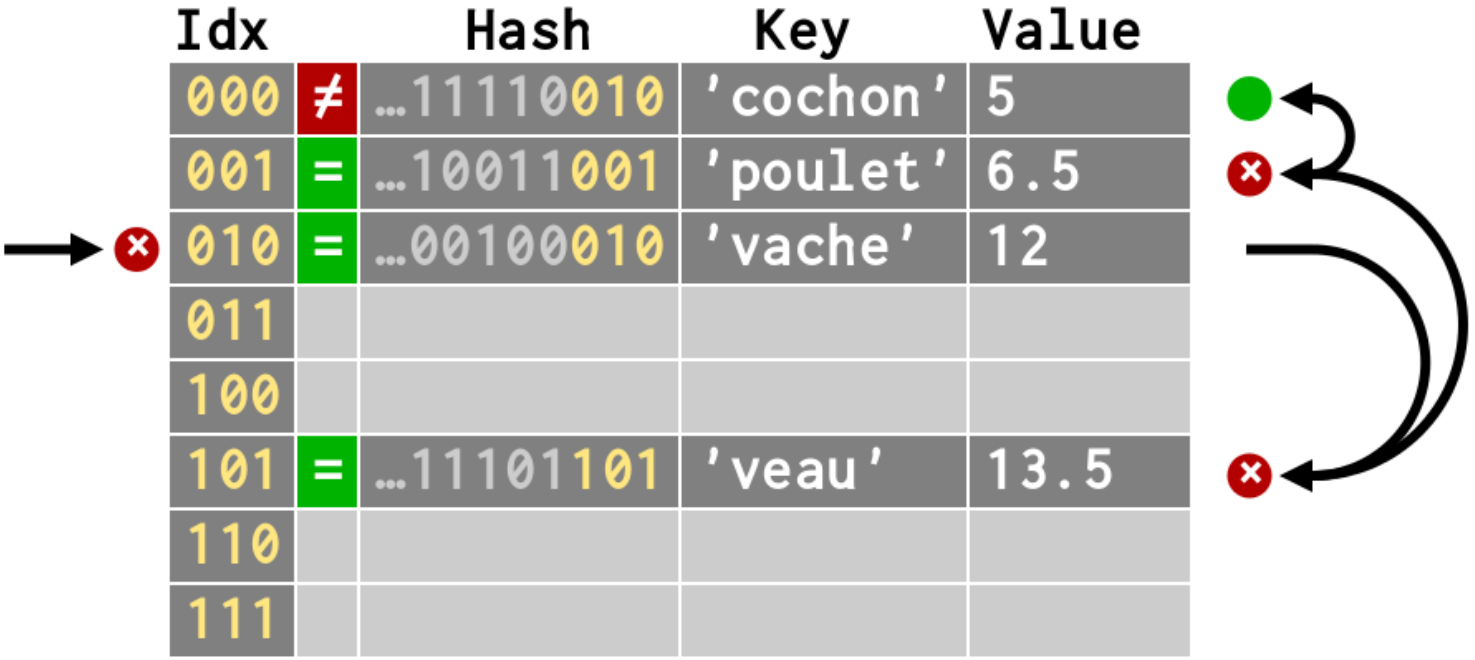


Idx		Hash	Key	Value
000				
001	=	…10011001	'poulet'	6.5
010	=	…00100010	'vache'	12
011				
100				
101	=	…11101101	'veau'	13.5
110				
111				

La table de hashcode ou le dictionnaire (2)

`animaux['cochon'] = 5`

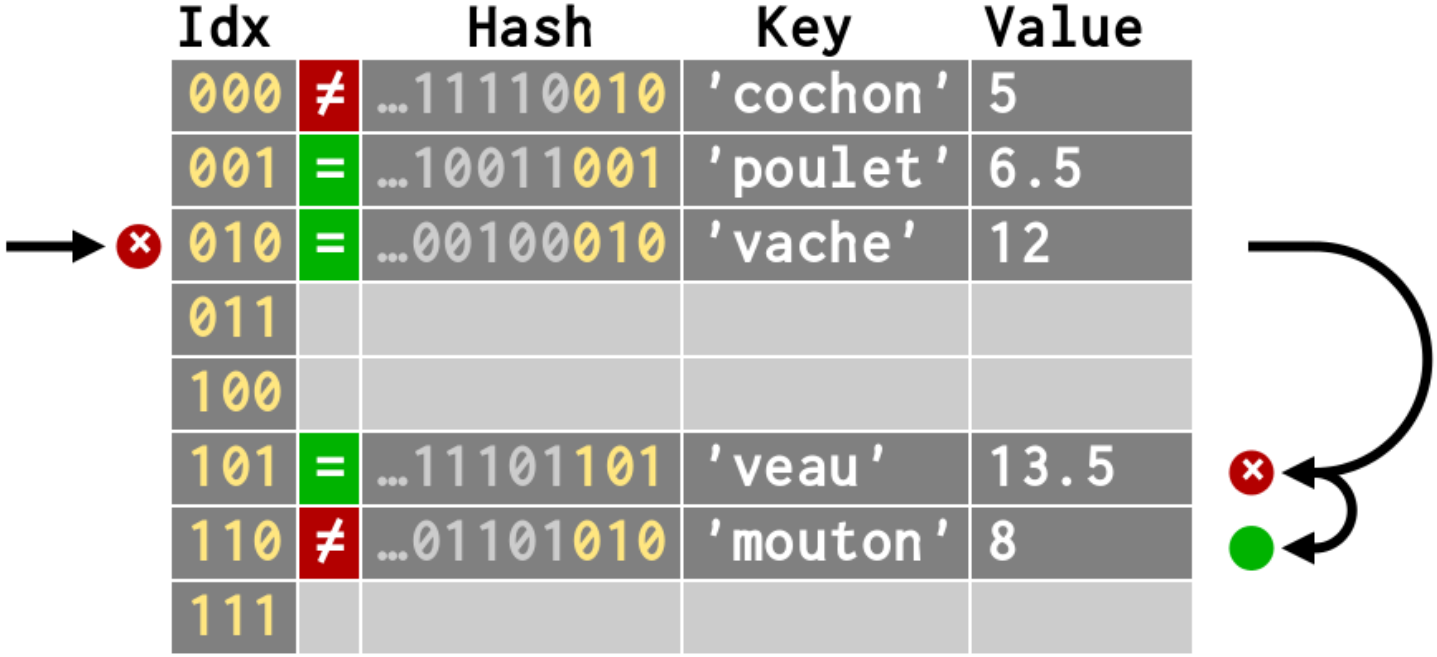
Il y a un conflit : on utilise alors une fonction secondaire (open adressing) pour trouver un emplacement libre – dans ce cas on essaye successivement [2, 5, 1, 0] (en général c'est mieux !)



La table de hashcode ou le dictionnaire (3)

`animaux['mouton'] = 8`

Il y a encore un conflit : dans ce cas on essaye successivement [2, 5, 6]



La table de hashcode ou le dictionnaire (4)

Pour trouver un élément dans la table : on refait exactement les mêmes opérations.

Quand la table est remplie au 2/3 alors on augmente la taille de la table. Si Il y a moins de 50000 clés, alors on la multiplie par 4 sinon par 2. La valeur initiale d'un dictionnaire vide est de 8 entrees.

Question : Est-ce que les conflits ne vont pas dégrader les performances ? Est-ce que le remplissage de la table est uniforme ? Pour pouvoir répondre à cette question il faut se plonger (un peu) dans les probabilités !

Le problème du remplissage

En anglais : ”Occupancy problem”. On veut placer au hasard M balles[les mots] dans N boites[les cases]. On cherche à savoir comment les différentes boites vont être remplies. De manière plus précise : soit X_i la variable aléatoire qui compte le nombre de balles dans la boite i. On souhaite connaître :

- La probabilité qu’il y ait k balles dans la boite i : $Pr[X_1 = k]$ (y compris pour $k = 0$)
- La probabilité pour qu’une boite contienne plus de k balles : $Pr[X_1 > k]$
- Le nombre moyen de tirages pour avoir une boite avec deux balles. C’est le problème de l’anniversaire
- Le nombre moyen de tirages pour remplir toutes les boites ”coupon collector”

La loi de probabilité

Cherchons maintenant la répartition dans chaque boite. Pour cela nous allons calculer quelle est la probabilité que la boite 1 contienne exactement k boules.
Si on considère la boite 1, pour chaque tirage d'une boule, on a une probabilité de $\frac{1}{N}$ que la boule soit dans la boite 1 et $1 - 1/N$ que la boule soit ailleurs que dans la boite 1. Il s'agit d'un schéma de Bernouilli (loi binomiale).

$$Pr[X_1 = k] = C_M^k \left(\frac{1}{N}\right)^k \left(1 - \frac{1}{N}\right)^{M-k}$$

On pose toujours $\alpha = M/N$.

La loi de probabilité(2)

Si $k = 0$ alors : $Pr[X_1 = 0] = \left(1 - \frac{1}{N}\right)^M$ si M et N sont grand alors : $\left(1 - \frac{1}{N}\right)^M = \left(\left(1 - \frac{1}{N}\right)^N\right)^{M/N} = e^{-\alpha}$. Donc premier résultat :

$$Pr[X_1 = 0] = e^{-\alpha}$$

De manière générale on trouve, si k est petit et M et N grand :

$$Pr[X_1 = k] = \frac{\alpha^k}{k!} e^{-\alpha}$$

Ce qui n'est rien d'autre qu'une loi de Poisson. (Problème des files d'attente, ...)

Vérification expérimentale (1)

Si on ne comprend pas bien les probabilités, on peut toujours faire des simulations en utilisant la fonction `random` de Python pour voir ce qui se passe
J'ai pris $M = 424579$ et $N = 1048576$, ce qui correspond à $\alpha = 0.405$. Et on lance les tirages ainsi :

vérification aléatoire

```
random.seed(111111)
table = [0] * N
for key in range(0,M) :
    table[random.randint(0,N - 1)] += 1
compteAffiche(table)
```

Vérification expérimentale (1a)

et on trouve :

k	nb de boites mesurée	loi de poisson
0	699149	699438
1	283705	283209
2	57195	57337
3	7698	7738
4	758	783
5	68	63
6	3	4

Les calculs précédents semblent donc justes !!! De cette manière on peut facilement appréhender des lois de probabilités. Par exemple on peut vérifier le ”coupon collector” : combien de tirage pour remplir toutes les boites (on devrait trouver $n \log n$).

Vérification expérimentale (2)

Maintenant, au lieu de faire un tirage aléatoire on va remplir les entrées de la table avec notre fonction de hashcode (celle de Python). On a bien sûr les mêmes paramètres que précédement.

simulation du hashcode

```
table = [0] * N
for key in listeMots :
    table[hash(key) % N] += 1
compteAffiche(table)
```

Cela simule la manière dont notre table de hashcode serait remplie.

Vérification expérimentale (2a)

k	nb de boites mesurée	loi de poisson
0	699487	699438
1	283038	283209
2	57498	57337
3	7735	7738
4	754	783
5	60	63
6	4	4

Et on trouve qu'avec notre dictionnaire qui n'a rien d'aléatoire, le remplissage de la table de hashcode correspond parfaitement à la loi de Poisson. Nous pouvons donc prendre l'hypothèse de la loi de Poisson pour modéliser le hashcode. En particulier l'essentiel des entrées est de taille 1 ou 2. Donc très peu de conflits : c'est pour cela que la hashcode marche bien !!

Un test en vrai grandeur

J'ai choisit un roman de Marcel Prost : Du coté de chez Swann (http ://www.gutenberg.org/files/2650/2650-0.txt). Ce roman a 181 518 mots.

Pour chacun des mots on consulte le dictionnaire pour savoir si le mot est correct. Pour le cas des mots qui commencent une phrase, je fais un deuxième essai en mettant le mot en minuscule.

Le temps de traitement est de 40 ms. Nos hypothèses sont donc raisonnables.

OUI, il y a des fautes :

- tout d'abord il y a un certain nombre de noms propres qui ne sont pas dans mon dictionnaire, par exemple « Swann ».
- ensuite il y a des mots incomplets, par exemple « jusqu ».
- il y a de vraies fautes, par exemple « psychiâtres » (erreur de transcription ?)
- il y a des mots qui ne sont pas dans mon dictionnaire, par exemple « dodonéenne » (relative à la ville de Dodone, dans l'Épire)

Un test en vrai grandeur (2)

Correction de Proust

```
expr = re.compile("\W+",re.U)
listeDesMots = [item  for ligne in open("DuCoteDeChezSwann.txt", 'r')
                      for item in expr.split(ligne) if item != '']
print("Nombre de mots du texte : ", len(listeDesMots))
debut = time.clock()
fautes = []
for mot in listeDesMots :
    if mot in dico : continue
    if mot.lower() in dico  : continue
    fautes.append(mot)
print("Temps d'exécution : ", "%.3f" % (time.clock() - debut), "s")

# On imprime les fautes qui ne sont pas des noms propres
print([mot for mot in fautes if mot[0] != mot[0].upper()])
```

(Montrer les résultats dans dico2.py)

Une bonne fonction de hachage

Si k est une chaîne $k = c_0 c_1 \dots c_{n-1}$ (c_i code numérique du $i$ème caractère de k). Il suffit de choisir un nombre a et de calculer avec le schéma de Horner :

$$h(k) = c_0 + a c_1 + a^2 c_2 + \cdots + a^{n-1} c_{n-1} \pmod{N}$$

le programme en Java

```
k = 0;
for (int i = n - 1; i >= 0; i--)
    k = k*a + c[i] % N;
```

Java a choisit $a = 31$ (Python 1000003 ??), vous pouvez prendre pour a n'importe quelle valeur, sauf une puissance de 2, si le codage d'un caractère est basé sur une puissance de 2.

Une bonne fonction de hachage (2)

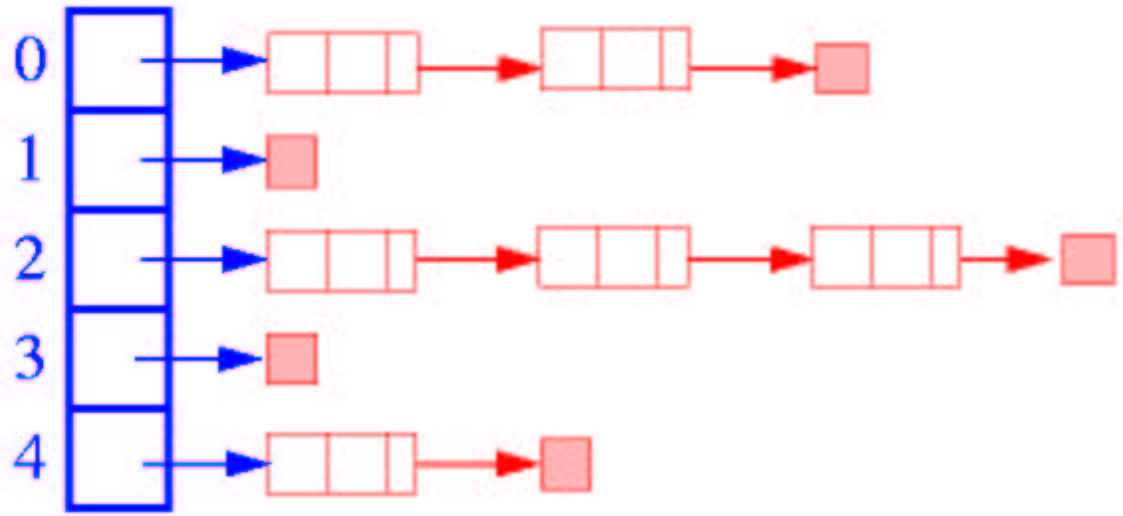
Mieux ? : fonction de hachage universelle.

le programme en Java

```
a = 31415 ; b = 27183 ;
k = 0 ;
for (int i = 0 ; i < n ; i++) {
    k = k*a + c[i] % N ;
    a = a*b %(N - 1) ;
}
```

Méthode A : Liste chaînées

Toutes les clés ayant le même hachage (conflit) sont chaînées entre-elles. Cherchons le nombre moyen d'éléments à comparer en cas de succès (le mot est dans le dictionnaire) et en cas d'échec.



Méthode A : Liste chaînées (2)

- En cas d'échec, c'est plus simple : il y a M mots dans le dictionnaire et N entrées. Soit le hachage n'est pas dans la table, soit il faut parcourir toute la liste chaînée jusqu'au bout. On a vu que la longueur moyenne d'une liste était de $\alpha = M/N$ donc le nombre moyen de comparaison est α
- En cas de succès : On peut admettre que l'on parcourt en moyenne la moitié de la liste. Comme cette longueur moyenne est α et qu'il faut toujours au moins une comparaison on trouve : nombre moyen de comparaisons est donc $1 + \alpha/2$
- Occupation mémoire : N mots pour la table + M mots pour les clés + M mots pour les liens. Ce qui donne en tout :

$$N + M + M = N + 2N\alpha = N(1 + 2\alpha)$$

Méthode B : Linear Probing

Dans ce cas on met les entrées directement dans la table. Quand il y a un conflit, alors on parcours la table circulairement et on met la nouvelle entrée dans la première case libre. Quand on recherche un élément on fait la même chose, si on tombe sur une case vide, alors l'élément n'est pas dans la table.

En java

```
public boolean containsKey(String s) {
    int i = hash(s);
    while (table[i] != null) {
        if (s.equals(table[i])) return true;
        i = (i + 1) % table.length;
    }
    return false;
}
```

Méthode B : Linear Probing (2)

Il se crée des ”clusters” et l’analyse est beaucoup plus complexe. Il faudrait connaître la répartition statistique des longueurs des cluster.

```
alpha = 0.57, N = 600 000, M = 342538
Nb de clusters = 104311, lg moy. d'un cluster = 3.284,
carré moyen de la taille d'un cluster = 4.706
taille  Nombre   taille  Nombre
  1      43258      8      1901
  2      20741      9      1485
  3      11989     10      1041
  4       7328     ....
  5       5026     73         3
  6       3732     74         0
  7       2608     75         1
```

Méthode B : Linear Probing (3)

Je ne sais pas démontrer ces formules. Mais voici les nombres de comparaisons moyens (dans ce cas $\alpha < 1$) :

- En cas de succès :

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- En cas d'échec :

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

- Occupation mémoire : N et c'est tout !!

Méthode C : Double Hashing – Open adressing

Comme la méthode précédente sauf que quand il y a un conflit, on ne cherche pas la prochaine case libre, mais on fait un second hachage et on cherche ainsi la prochaine case libre. Le but est de diminuer la taille des clusters. C'est la méthode choisie par Python.

En java

```
public boolean containsKey(String s) {
    int i = hash(s);
    while (table[i] != null) {
        if (s.equals(table[i])) return true;
        i = (i + hash2(s)) % table.length;
    }
    return false;
}
```

Méthode C : Double Hashing (2)

Voici les formules :

- En cas de succès :

$$\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

- En cas d'échec :

$$\frac{1}{1-\alpha}$$

- Occupation mémoire : N et c'est tout !!

Qui est le meilleur ?

Si on veut faire des comparaisons équitables, il faut se mettre dans les mêmes conditions au niveau de l'occupation mémoire. Par exemple si l'on choisit dans le cas du chaînage d'avoir une mémoire totale de 900 000 mots on trouve alors avec M = 342538 : N = 214924 et $\alpha = 1.59$.

Méthode	α	lgSu	lgEch
SeparateChaining	1.59	1.708	1.592
LinearProbing	0.381	1.326	1.109
DoubleHashing	0.381	1.203	0.712

Donc double hashing est la meilleure méthode sauf si le temps du double hachage est trop long !!

Ceci est un exemple d'algorithme ou on échange la taille mémoire avec la performance – Remarquez que maintenant une taille mémoire de 4Go est courante et souvent peu utilisée

Un dernier exemple

A chaque entrée dans un dictionnaire, on peut affecter n'importe quelle structure Python, en particulier un autre dictionnaire.

Un exemple

```
eleves = {
    'A' : {'Math' :12, 'Physique' : 14, 'Anglais' :13, 'Option1' :5},
    'B' : {'Math' :13, 'Physique' : 13, 'Anglais' :13, 'Option2' :6},
    'C' : {'Math' :14, 'Physique' : 12, 'Anglais' :13, 'Option1' :7},
    'D' : {'Math' :15, 'Physique' : 11, 'Anglais' :13, 'Option3' :8},
}
```

et on obtient alors facilement la note d'un élève pour une matière ainsi `eleve['C']['Physique']`, bien sûr si les champs existent. Sinon c'est un tout petit peu plus compliqué.

Un dernier exemple (2)

Cette exemple vous montre que grâce au dictionnaire, il est facile de construire dans un programme des structures qui rendent les mêmes services qu'une petite base de données (quelques 100 000 éléments) et avec la même efficacité.

- Cette structure (clé,valeur) correspond a ce que l'on trouve maintenant dans ce que l'on appelle les bases de données NOSQL.
- Souvenez vous des structures de fichier "json" que l'on avait vu l'année dernière : il n'y a que des listes et des dictionnaires.
- Il est alors facile d'avoir sa base de données sous un format "json" qu'il est facile de lire et d'écrire et vous avez tous les traitements Python à votre disposition.

Le Devoir - Correcteur orthographique

Ceci fera l'objet du prochain TP : le Vendredi 15 Février

Il ne s'agit que de corriger les fautes d'orthographes. C'est à dire que si le mot n'appartient pas au dictionnaire, il y a une faute. Le but est donc : étant donné un mot qui n'est pas dans un dictionnaire, de trouver les mots les plus *proches* dans ce dictionnaire.

Il y a beaucoup de méthodes. La méthode que je vous propose utilise les trigrammes. Il y a 2 étapes :

① la construction du dictionnaire des tri-grammes

② La correction orthographique d'un mot

Le Devoir - La distance de Levenshtein

C'est la distance naturelle de comparaison de 2 mots. On va considérer les opérations de remplacement d'un caractère par un autre, de suppression d'un caractère et d'insertion d'un caractère. Cette distance entre 2 mots est le nombre minimal d'opération pour passer d'un mot à un autre. Par exemple la distance entre omnibulé et obnubilé (le bon) est de 3 :

1. omnibulé
2. obnibulé - remplacement m par b
3. obnubulé - remplacement i par u
4. obnubilé - remplacement de u par i

On va voir cet algorithme qui se programme avec la programmation dynamique.

La distance de Levenshtein

L'idée de la programmation dynamique consiste à trouver la solution optimale petit à petit. On cherche $d(s,t)$. Dans notre cas, supposons que l'on connaisse la distance $d(s_i,t_j)$ (ou s_i représente les i premiers caractères de s et t_j les j premiers caractères de t). On peut alors établir l'équation de récurrence suivante :

Si $s[i] = t[j]$ alors $d(s_i,t_j) = d(s_{i-1},t_{j-1})$

Sinon $d(s_i,t_j) = 1 + \min(d(s_{i-1},t_{j-1}), d(s_{i-1},t_j), d(s_i,t_{j-1}))$

$d(.,t_j) = j$

$d(s_i,.) = i$

Le résultat se trouvant dans $d(s_m,t_n)$ si m et n sont respectivement les longueurs de s et t.

La distance de Levenshtein(2)

On va calculer la distance entre chiens et niche

	.	n	i	c	h	e
.	0	1	2	3	4	5
c	1	1	2	2	3	4
h	2	2	2	3	2	3
i	3	3	2	3	3	3
e	4	4	3	3	4	3
n	5	4	4	4	4	4
s	6	5	5	5	5	5

Cela correspond à l'alignement suivant :

```
--chiens
nich-e--
```

La distance de Levenshtein(3)

Le programme

```
def levenshtein(s, t) :
    m = len(s)
    n = len(t)
    L = [[0] * (n + 1) for x in range(m + 1)]
    # il y a une case vide au debut de chaque chaine
    for i in range(m + 1) : L[i][0] = i
    for j in range(n + 1) : L[0][j] = j
    for i in range(1, m + 1) :
        for j in range(1, n + 1) :
            if s[i - 1] == t[j - 1] :
                L[i][j] = L[i - 1][j - 1]
            else :
                L[i][j] = 1 + min(L[i - 1][j], # deletion
                                  L[i][j - 1], # insertion
                                  L[i - 1][j - 1]) # substitution
    return L[m][n]
```

Le correcteur orthographique

L'idée la plus simple : quand un mot n'appartient pas au dictionnaire (mal orthographié), alors on calcule sa distance de Levenshtein avec tous les mots du dictionnaire.

Les mots les plus proches seront alors proposés comme correction.

Cependant cette méthode n'est pas très efficace : le calcul de 400000 distances est assez long (le mesurer). Je vais donc vous proposer une autre méthode qui aboutit a peu près au même résultat

La construction du dictionnaire

On lit une liste de mots corrects (le dictionnaire). Pour chaque mot

1. on rajoute un caractère « $ » au début et à la fin du mot. « accueil » devient « $accueil$ »
2. Puis on construit la liste des tri-grammes apparaissant dans ce mot. « $ac », « acc », « ccu », « cue », « uei », « eil », « il$ » .
3. On construit alors au fur et à mesure le dictionnaire des tri-grammes. Chaque tri-gramme pointant vers la liste des mots contenant ce tri-gramme. Par exemple « acc » pointe vers la liste « accueil », « accent », « accélération », « raccourci », ….

La correction d'un mot

1. on rajoute un caractère « $ » au début et à la fin du mot. « acceuil » devient « $acceuil$ »
2. Puis on construit la liste des tri-grammes apparaissant dans ce mot. « $ac », « acc », « cce », « ceu », « eui », « uil », « il$ » .
3. En consultant notre liste de tri-grammes, on recherche les mots (corrects) du dictionnaire contenant le plus possible de ces tri-grammes. Par exemple « accueil » contient 3 tri-grammes en commun avec « acceuil », mais « fauteuil » contient aussi 3 tri-grammes commun avec « acceuil ».
4. On va trier cette liste de candidats à la correction en fonction du coefficient de Jaccard. On ne va garder que les candidats ayant un coefficient > 0,2.
5. Parmi ces candidats potentiels, on va les trier en utilisant la distance de Levenshtein. On va proposer comme liste de correction, les 5 premiers de cette liste par exemple.

Le coefficient de Jaccard

Quand on cherche les mots contenant le plus de triplets en commun avec un mot faux, on trouve souvent des mots beaucoup plus grands – qui ne sont donc pas bon.

Ce coefficient est un moyen simple de mesurer la similarité de deux ensembles A et B.

$$c = \frac{|A \cap B|}{|A \cup B|}$$

ce qui se traduit dans notre cas par (si M est le mot faux et si D est un mot du dictionnaire) :

$$c = \frac{nbTrigrammesCommuns(D)}{|M| + |D| - nbTrigrammesCommuns(D)}$$

Le nombre de trigramme d'un mot est $|mot| + 2 - 2$