Chapitre 2

DICTIONNAIRES ET FONCTIONS DE HACHAGE

Table des matières

Cette première partie a pour objet l'étude d'une structure de données que vous avez déjà rencontré en première année : les dictionnaires. Cette structure de données répond à la problématique suivante : comment rechercher efficacement de l'information dans un ensemble géré de façon dynamique (c'est à dire dont le contenu est susceptible d'évoluer au cours du temps).

Les dictionnaires sont couramment utilisées en informatique : c'est par exemple la structure de données utilisée pour gérer les systèmes de noms de domaine (DNS, pour Domain Name System). Les ordinateurs connectés à un réseau comme Internet possèdent une adresse numérique (en IPv4 par exemple, celles-ci sont représentées sous la forme xxx.xxx.xxx.xxx, où xxx est un nombre hexadécimal variant entre 0 et 255). Pour faciliter l'accès aux systèmes qui disposent de ces adresses, un mécanisme a été mis en place pour associer un nom (plus facile à retenir) à une adresse IP. Ce mécanisme utilise un dictionnaire dans laquelle les clefs sont les noms de domaine et les valeurs les adresses IP.

I DICTIONNAIRES : RAPPELS

Type de variable	Syntaxe Python	Description	Exemple
Dictionnaire	`dict`	collections d'éléments avec clés et valeurs	`{clef:valeur,...}`

▶ Un **dictionnaire** (type `dict`) est une collection d'**éléments** composés d'une **clé** associée à une **valeur**.

▶ Contrairement aux listes qui sont délimitées par des crochets, on utilise des **accolades** pour les dictionnaires.

▶ Un dictionnaire en Python va permettre de rassembler des éléments (comme des listes ou tuples) mais ceux-ci seront **identifiés par une clé**. *Analogie à un dictionnaire où on accède à une définition avec un mot.*

▶ **Commandes utiles pour un dictionnaire :**

Méthode 1 - Directement par la liste de ses éléments :
(par exemple le nombre de roues d'un type de véhicule) :

```
dico = {"voiture": 4, "vélo": 2, "tricycle": 3}
```

Remarque.
- Ici les clés sont `"voiture"`, `"vélo"` et `"tricycle"`.
- Les valeurs sont les entiers `4`, `2` et `3`.
- Il n'y a **pas d'ordre** entre les clefs ! (clefs non numérotées)
- **Les clefs peuvent être n'importe quel type de variable sauf des listes ou dictionnaires.**

Méthode 2 - Ajout un par un des éléments

```
dico = {}

dico["voiture"]=4
dico["vélo"]=2
dico["tricycle"]=3
```

Remarque.
- Si une clé n'existe pas dans le dictionnaire **elle est créé avec la valeur associée**.
- Si une clé existe déjà dans le dictionnaire, **sa valeur est remplacée**.

Exemple. Exemple de dictionnaire avec différents types de variables :

```
dico = {6:"petit" , "violet":True , False:[1,2,3] }
```

Obtenir les clés :

```
for key in dico:
    print(key)
```

Obtenir les valeurs :

```
for key in dico:
    print(dico[key])
```

Commandes	Description
`val=dico[key]`	Stocke dans `val` la valeur associée à la clé `key` dans `dico`
`dico[key]=val`	Remplace la valeur associée à la clé `key` par `val` dans `dico` (si `key` était déjà présente)
`dico[key]=val`	Ajoute la clé `key` et sa valeur associée `val` dans `dico` (si `key` n'était pas présente)
`if key in dico:`	Teste si la clé `key` est dans `dico` (*complexité en* $\mathcal{O}(1)$)
`if dico[key]==val:`	Teste si la valeur associée à la clé `key` est égale à `val`

COMPARATIF LISTES/TUPLES/CHAÎNES/DICTIONNAIRES

Le tableau ci-dessous regroupe les principales spécificités et différences entre ces 4 types de variables.

On utilise le plus adapté en fonction de ce qui est demandé.

Tableau	Liste	Tuple	Chaine de caractères	Dictionnaire
type `list`	type `list`	type `tuple`	type `str`	type `dict`
`["a",True,7]`	`[3.14,2.71,1,41]`	`("a",True,7)`	`"Essouriau"`	`{ clé : valeur , ... }`
modifiable	modifiable	**non modifiable**	**non modifiable**	modifiable
ordonné	ordonné	ordonné	ordonné	**non ordonné**

II DICTIONNAIRES : COMPLÉMENTS

II.1 DÉFINITION D'UN DICTIONNAIRE PAR COMPRÉHENSION

A partir de deux listes on peut créer un dictionnaire par compréhension.

Exemple. Avec `clefs = [5,2,25]` et `valeurs = ["pommes","poires","fraises"]` on peut créer :
`D={clefs[i]:valeurs[i] for i in range(len(clefs))]}`

II.2 ACCÈS AUX CLEFS, VALEURS, ITEMS

Les commandes `D.keys()`, `D.values()` et `D.items()` renvoient respectivement la collection des clés, des valeurs et des couples (clé, valeurs). Attention ce ne sont pas des listes, on peut les convertir en liste si besoin grâce aux commandes `list(D.keys())`, `list(D.values())` et `list(D.items())`.

On peut quand même écrire une boucle sur l'ensemble des clés, valeurs ou items d'un dictionnaire :

```
for cle in D.keys():
    print(cle)
    print(D[cle])

for cle in D:
    print(cle)
    print(D[cle])
```

```
for valeur in D.values():
    print(valeur)

for item in D.items():
    print(item)
```

Remarque. A noter que `len(D)` donne aussi le nombre de couples (clé,valeur) dans le dictionnaire.

Remarque. La commande `del D[c]` supprime une association si la clef est présente dans le dictionnaire. Attention, elle déclenche l'exception `KeyError` sinon.

II.3 COPIE D'UN DICTIONNAIRE

▶ **Copie superficielle et en profondeur de listes**

- ◇ Si `L` est une liste et si `M=L`, alors une modification de `M` entraîne une modification de `L` (et vice-et-versa). On parle de **copie superficielle**.
  Attention, même une fonction modifie une liste même sans `return`, même s'il cette liste est modifiée en tant que variable locale. C'est ce qu'on appelle **l'effet de bord**.
- ◇ Si vous voulez faire évoluer `L` et `M` de façon indépendante, la commande `M=L.copy()` pare le problème. On parle de **copie en profondeur**.

Si jamais vous avez oublié la commande `copy` ce n'est pas grave : `M=L.copy()` ⇔ `M=[l for l in L]` ⇔ `M=L[:]`

Exemple. A noter que cette parade ne fonctionne pas si les éléments de `L` sont eux-même des listes :

```
L=[1,2,3]
M=L
M[2]=5
```

`L[2]` vaut alors 5.

```
L=[1,2,3]
M=L.copy()
M[2]=5
```

`L[2]` vaut encore 3.

```
L=[[1,2,4],2,3]
M=L.copy()
M[0][2]=5
```

`L[0]` vaut alors `[1,2,5]`.

Remarque. Pour éviter cela grâce à une copie profonde avec des commandes qui ne sont pas au programme.

▶ **Copie superficielle et en profondeur de dictionnaires**

Pour les dictionnaires, c'est le même problème et `copy` fonctionne de la même manière que pour les listes :

```
D={"a":1,"b":2,"c":3}
Dp=D
Dp["b"]=5
```

`D["b"]` vaut alors 5.

```
D={"a":1,"b":2,"c":3}
Dp=D.copy()
Dp["b"]=5
```

`D["b"]` vaut encore 2.

II.4 TYPE DE VARIABLE POUVANT ÊTRE DES CLEFS

▶ Les **valeurs** d'un dictionnaire peuvent être de tous types : entiers, flottants, listes, tuples et même des dictionnaires.

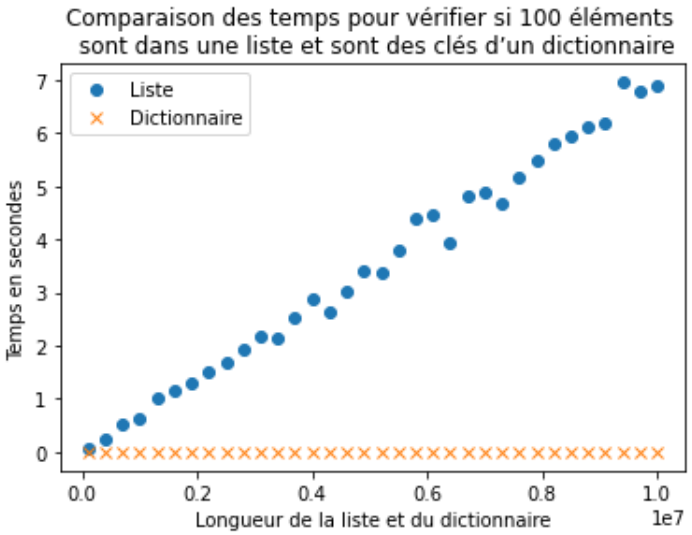
▶ En revanche, les **clés** ne peuvent pas être les listes ou des dictionnaires, dans le cas des tuples il ne faut pas non plus que les éléments de ces tuples puissent être eux-même des listes ou des dictionnaires.
Nous verrons plus tard pourquoi c'est le cas.

III IMPLÉMENTATION DES DICTIONNAIRES : HACHAGE

III.1 UNE RECHERCHE D'ÉLÉMENT PLUS RAPIDE QUE POUR LES LISTES

Plus une liste est grande plus il faudra de temps pour vérifier si un élément donné est dans cette liste. Ce n'est pas le cas pour les dictionnaires, ou le temps pour vérifier si un élément est une clé d'un dictionnaire est quasi-constant.

La figure ci-contre illustre ce principe [a] : Pour les listes, c'est assez facile à comprendre si un élément n'est pas dans une liste, pour le vérifier il va falloir regarder les éléments un à un de la liste et si une liste contient mille fois plus d'éléments qu'une autre, cela fera mécaniquement mille fois plus de vérifications [b].



a. Vous pourrez retrouver le script sur le Cahier de Prépa de PSI du lycée de l'Essouriau.

b. On rappelle que lorsqu'on réfléchit en terme de complexité, on regarde souvent le cas le plus défavorable.

On rappelle l'algorithme qui vérifie si un élément est dans une liste ci-contre. Il est logique que la complexité pour savoir si un élément est dans une liste soit linéaire en la longueur de la liste. Néanmoins cela ne semble pas être le cas pour les dictionnaires. (Rappelons qu'il n'y a pas d'ordre dans les clés ou dans les valeurs.)

```
1 def Recherche(L:list,x)->bool:
2   for i in range(len(L)):
3     if L[i]==x:
4       return True
5   return False
```

Le temps nécessaire pour savoir si un élément est une clé d'un dictionnaire semble constant. Ceci est liée à l'implémentation des dictionnaires qui diffère de celui des listes.

On peut toujours implémenter un dictionnaire en utilisant simplement une liste de couples `(key,elt)`. Les opérations précédentes (à part la création) seront de complexité linéaire en le nombre de couples stockés dans le dictionnaire (en supposant que les clés sont de taille bornée pour que le test d'égalité s'effectue en temps constant). Dans la suite, on veut faire mieux !

III.2 PRINCIPES ET LIMITES DE L'ADRESSAGE DIRECT

Supposons que l'on sache que les clés à stocker soient des entiers de $K = [\![0, m[\![$, avec m un entier pas trop grand. Il est alors possible de réaliser la structure de dictionnaire dans une liste `L` de taille m. En supposant qu'un élément associé à une clé ne puisse jamais être `None`, on pourra réaliser le dictionnaire avec la convention :

$$\text{Pour } \texttt{k} \in [\![0, m-1]\!] \text{ , } \texttt{L[k]} = \begin{cases} \texttt{None} & \text{si la clé } \texttt{k} \text{ n'est pas présente ;} \\ \text{l'élément } \texttt{e} \text{ associé} & \text{sinon.} \end{cases}$$

Voici par exemple un dictionnaire dont on sait que les clefs ne peuvent être que des entiers entre 0 et 9, Le dictionnaire contient actuellement 3 couples : `(1,'abc')`, `(5,54)` et `(7,True)`.

```
[None, 'abc', None, None, None, 54, None, True, None, None]
```

Il est facile d'implémenter toutes les opérations de la liste ci-dessus avec cette structure, et elles s'effectuent toutes en temps constant, sauf la création ! En effet, l'opération de création (qui devrait prendre en paramètre l'entier m bornant les clés possibles) s'effectue en temps $O(m)$.

L'adressage direct, outre le fait qu'il impose une contrainte forte sur les clés (nécessairement des entiers positifs pas trop grands), demande également une complexité mémoire $O(m)$ quel que soit le nombre de clés effectivement contenues dans le dictionnaire. Il est préférable d'avoir une complexité mémoire linéaire en ce nombre, pour ne pas gaspiller de la mémoire.

III.3 TABLES/FONCTIONS DE HACHAGE

▶ But des fonctions de hachage

L'idée des **fonctions de hachage** est de ramener l'univers des clés possibles vers un ensemble fini $[\![0, m-1]\!]$, avec m pas trop grand, avec une fonction h transformant la clef en entier dans $[\![0, m-1]\!]$.

On utilisera alors une liste de taille m pour stocker les couples (clé,élément). Plus précisément, avec `L` une telle liste, `L[i]` sera la liste des couples de clés dont l'image par la fonction de hachage est `i`.

Comme une telle fonction de hachage n'est en général pas injective, il se peut que deux clés k et k' aient la même image via la fonction de hachage h. On appelle m la **largeur de hachage**.

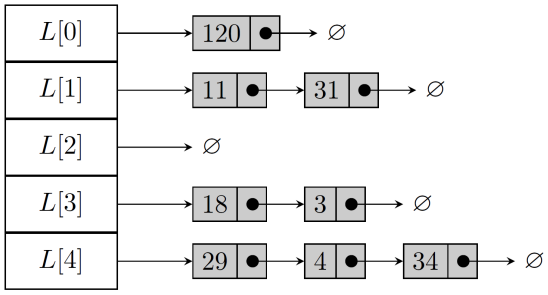


Illustration d'une table de hachage de largeur $m = 5$ pour la liste :
`L = [[120], [11, 31], [], [18, 3], [29, 4, 34]]`

▶ Définition et exemples de fonction de hachage

Une **fonction de hachage** h est une fonction, qui, à partir d'une donnée fournie en entrée, renvoie un nombre de taille fixe. Dans une utilisation en cryptographie, une fonction de hachage doit vérifier les propriétés suivantes :

- le résultat **se calcule rapidement** et **paraît aléatoire** : modifier un tout petit peu l'entrée modifie énormément la sortie. (processus déterministe : le hachage a le même résultat sur la même entrée !)
- il est très **difficile de construire des collisions** : c'est-à-dire de trouver deux entrées différentes dont l'image par la fonction de hachage est la même.

Exemple. On peut citer MD52 (obsolète), la famille SHA-2 3, et bien d'autres...
Python propose une fonction de hachage avec la commande `__hash__()`, l'entier renvoyé est un entier sur 64 bits en complément à deux (donc dans $[\![-2^{63}, 2^{63}-1]\!]$)

```
In[1]: hash("Lycée␣de␣l'Essouriau")
Out[1]: 6587211257724051 33
```

Remarque.
Tous les types de variable ne sont pas « **hachables** » (*on peut leur appliquer la fonction de hachage*).
- Les entiers, flottants, chaînes de caractères, tuples sont hachages (car immuables).
- Les listes et dictionnaires ne sont pas hachages (modifiables + effet de bord notamment).

Donc les tuples ne doivent pas contenir de listes et dictionnaires !

C'est la raison pour laquelle les listes et les dictionnaires ne peuvent être des clefs dans un dictionnaire.

```
In[2]: hash(np.pi)
Out[2]: 326490430436040707
```

```
In[3]: hash((2023,"PCSI"))
Out[3]: -7544970212294543536
```

```
In[4]: hash([1,2,3])
TypeError: unhashable type
```

III.4 UTILISATION POUR LA STRUCTURE D'UN DICTIONNAIRE

On fixe une largeur de hachage m, on peut forcer une fonction de hachage à prendre des valeurs entre 0 et $m-1$ en utilisant simplement le modulo :

```
def hachage(x):
  return hash(x)%m
```

On note h cette fonction de hachage à valeurs dans $[\![0, m-1]\!]$.

- Lorsqu'un dictionnaire est crée avec des clés $c_1, c_2, \dots, c_r$, Python va calculer $h(c_1), h(c_2), \dots, h(c_r)$.
- Python crée aussi un tableau de taille m, ce tableau est appelé **table de hachage**.
- Les éléments de ce tableau sont appelés **alvéoles**.
- Dans l'alvéole s, Python va stocker tous les couples (c_i, v_i) (où v_i est la valeur associée à la clé c_i) sous forme de liste tels que $h(c_i) = s$.

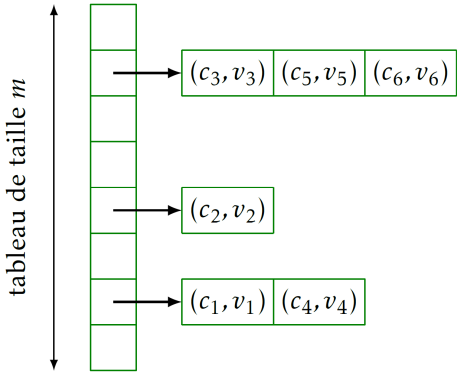


Table de hachage avec des collisions

Nous avons maintenant les outils pour comprendre pourquoi savoir si un élément est une clé d'un dictionnaire est si rapide. Si on veut savoir si x est une clé du dictionnaire, alors Python calcule $h(x)$.

Ainsi, le couple `(x,D[x])` devrait se trouver dans l'alvéole numéro $h(x)$ si `x` est une clé du dictionnaire, plusieurs cas peuvent se produire :

- l'alvéole $h(x)$ est vide, dans ce cas, il est immédiat que x n'est pas une clé.
- l'alvéole $h(x)$ n'est pas vide, dans ce cas Python va parcourir la liste dans l'alvéole en passant un à un tous les couples et regarder si le premier élément est x.
  Soit x est bien une clé et dans ce cas il va la trouver, soit non.

Le cas numéro 1 est bien sûr réglé assez rapidement, le cas numéro 2 peut poser plus de problème, en effet plus l'alvéole contient de couples (c_i, v_i) plus cela va prendre du temps. Ainsi, tout dépend de la capacité de la fonction de hachage à minimiser les collisions.

Ainsi, une bonne table de hachage aura tendance à répartir le plus uniformément possible les couples (clé, valeurs) dans les alvéoles.

Dans le pire cas possible, toutes les alvéoles sont vides, sauf une seule qui contient tous les couples (clé,valeurs). Ainsi, si on veut vérifier que x est une clé, il faudra passer en revue toutes les clés du dictionnaire, ceci prendra donc autant de temps que pour vérifier si un élément est une liste. Heureusement ce cas est hautement improbable.

► **Complexité : discussion**

En supposant que l'on travaille sur des objets de taille bornée, la complexité des opérations précédentes (sauf la création) est linéaire en la taille de l'alvéole numéro $i = h(k)$, où k est la clé impliquée.

Si on suppose que la fonction de hachage a tendance à bien répartir les clés entre alvéoles, on peut imaginer que chacune d'elle a une taille environ $\frac{n}{m}$, où n est le nombre de couples stockés et m la largeur de hachage.

Sans rentrer dans les détails probabilistes qui dépasseraient largement le cadre de ce cours, on admet que la complexité est de $O(1 + \frac{n}{m})$ pour les opérations précédentes (sans la création en $O(m)$), sous des hypothèses naturelles.

On a vu que les opérations de dictionnaire pouvait être considérée comme en $O(1)$, ce qui ne peut clairement pas être atteint avec la stratégie précédente si m est fixe. Il suffit pour cela de faire varier la largeur de la table, de manière à avoir toujours $\frac{n}{m}$ borné, avec n le nombre d'entrées stockées dans le dictionnaire.

IV EXERCICES

Exercice 1 (**Création**)**.**

Écrire une fonction `creation` qui prend en paramètre un entier n et qui renvoie un dictionnaire dont les clés sont les entiers de 1 à n, la valeur correspondant à une clé étant son opposé.
Par exemple `création(3)` vaut `{1:-1, 2:-2, 3:-3}`.

Exercice 2 (**min_max**)**.**

Écrire une fonction `min_max` qui prend en paramètre une liste de nombres non vide et qui renvoie un dictionnaire dont les clés sont les chaînes `"min"` et `"max"` avec pour valeurs respectives le minimum et le maximum des nombres de la liste. Par exemple `min_max([8,45,-32,135,5])` vaut `{"min":-32, "max":135}`.

Exercice 3 (**Nombre d'occurrences et comparaison de listes**)**.**

1. Écrire une fonction `Occurrence(L,x)` qui détermine le nombre d'occurrences d'une variable `x` dans une liste `L`.
2. En déduire une fonction `Occurrences(L)` utilisant obligatoirement la précédente qui renvoie un dictionnaire dont les clefs sont les éléments de `L` et dont les valeurs associées sont le nombre d'occurrences de ceux-ci dans `L`. Par exemple `occurrences([5, 42, -3, 5, 5, -3])` vaut `{5:3, 42:1, -3:2}`.
3. On note n la longueur de la liste `L`. Quelle est la complexité de la fonction précédente ? Écrire une nouvelle fonction qui fait exactement la même chose de complexité linéaire.
4. Écrire une fonction `taille` qui prend en argument un dictionnaire obtenu comme ci-dessus et qui renvoie la longueur de la liste à partir de laquelle le dictionnaire a été construit.
5. Écrire une fonction `compare` qui prend en argument deux listes et qui teste si les listes sont des permutations l'une de l'autre (elles contiennent les mêmes éléments le même nombre de fois, mais pas forcément dans le même ordre).
6. Quelle est la complexité de la fonction `compare` ?

Exercice 4 (**Température**)**.**

On dispose de noms de villes avec les températures moyennes relevées à une date donnée, enregistrées dans une liste de listes comme `[["Paris",12],["Lyon",14],["Marseille",19], ...]`.

1. Quelle est la complexité de chacune des opérations suivantes ?
   — accéder à la température d'une ville

— modifier la température d'une ville

— ajouter une nouvelle ville.

2. Afin de trouver la température d'une ville, on envisage de trier la liste suivant l'ordre alphabétique des noms de villes, puis effectuer une recherche dichotomique du nom de la ville. Quelle serait la complexité de cette recherche de température ?
3. Écrire une fonction `dico` qui prend en argument la liste de listes des villes et températures, et qui renvoie un dictionnaire dont les clés sont les noms des villes et les valeurs les températures.
4. Quelle est la complexité de la fonction `dico` ? Avec le dictionnaire obtenu, quelle est maintenant la complexité des opérations listées à la question 1 ? Conclure.
5. Écrire une fonction `température` qui prend en argument un dictionnaire comme ci-dessus et un nom de ville, et qui renvoie la température correspondant à la ville.

Pour les exercices suivants, si ce n'est pas déjà fait, veuillez télécharger le fichier `ListeMotsFrancais.txt` qui se trouve le Cahier de Prépa de la PSI de l'Essouriau.

On rappelle qu'il doit être mis dans le dossier où est votre fichier Python actuel.

Ce fichier contient tous les mots français. (Avec toutes les variantes : tous les accords possibles des adjectifs et des noms, toutes les conjugaisons des verbes. Les accents et les majuscules ont été retirés.)

On rappelle les commandes suivantes permettent d'ouvrir le fichier, lire son contenu et créer une variable `Liste` qui va contenir tous les mots (liste dont l'élément n°i est une chaine de caractères comportant les caractères écrits à la ligne n°i du fichier.

```
fichier=open("ListeMotsFrancais.txt", "r")
Liste=fichier.readlines()
```

Exercice 5 (**Dictionnaire des anagrammes**)**.**

1. Combien y-a-t-il de mots dans la liste ? (*Il y a* 323574 *mots dans la liste.*)
2. Attribuer à une variable, notée `mot`, le 50-ième mot de la liste et afficher sa valeur.
   (*Le cinquantième mot de la liste est : abajoues*)
3. Enlever le[1] caractère spécial `"\n"` à la fin de chaque mot dans `Liste`.
4. Tester les commandes `sorted(mot)` puis `"".join(sorted(mot))`.

   On rappelle qu'un mot est une anagramme d'un autre mot si on peut passer de l'un à l'autre en échangeant les lettres. Par exemple *chien* et *niche*.

   Remarquez alors que `"".join(sorted("chien"))` est égale à `"".join(sorted("niche"))`
5. Trouver les mots qui ont le plus d'anagrammes.

   Pour cela, en parcourant toute la liste, créer un dictionnaire dont les clés seront les chaînes de caractères de mots triés par ordre alphabétique et dont les valeurs seront le nombre de mots qui contiennent exactement les mêmes lettres mais dans un ordre différent.

   Par exemple, `D["cehin"]` vaudra 3, en effet, `chien`, `niche` et `chine` sont les seuls trois mots du dictionnaire qui contiennent une et une seule fois les lettres `c`, `e`, `h`, `i` et `n`.
6. Trouver la valeur maximale dans le dictionnaire précédemment crée ainsi que la ou les clés qui ont cette valeur maximale.
7. De plus, afficher tous les mots de `Liste` qui ont le nombre maximal d'anagrammes.

1. Le et non pas les, car contrairement aux apparences, `"\n"` compte pour un caractère et non deux caractères.

Exercice 6 (**Création d'une fonction de hachage**)**.**

On souhaite ici créer une fonction de hachage très simple (bien plus simple que celle utilisée par Python).

1. La commande `ord("z")` renvoie un entier naturel correspondant à un numéro pour la lettre `"z"` et de même pour toutes les lettres. Remarquez que `ord("x")-97` donnera un numéro à la lettre `x` entre 0 et 25. Créer une fonction `f(mot)` qui à un `mot` de longueur n va renvoyer :
$$\sum_{i=0}^{n-1}(\texttt{ord(mot[i])} - 97) \times 26^i$$
Vérifier que `f("cpge")` vaut 74752.
Vérifier que `f("anticonstitutionnellement")` vaut 177640564276178705646208861949989598.
La fonction `f` ferait-elle une bonne fonction de hachage ?
2. Créer une fonction de hachage `h1(mot)` qui a un mot renvoie le reste de la division euclidienne de `f(mot)` par $n = 10^6$.
3. Ainsi, si on créait des dictionnaires avec cette fonction de hachage, quelle serait la taille de la table de hachage ? Si on hache tous les mots du dictionnaire, y-aurait-il à votre avis des collisions ? Combien y-aurait-il en moyenne de mots dans chaque alvéole ?
4. Créer une fonction `Bilan` qui va construire une liste `L` à n éléments, tels que `L[i]` compte le nombre de mots de la liste `Liste` tels que `h1(mot)=i`. C'est-à-dire que `L[i]` va compter combien d'éléments serait dans l'alvéole `i` si on utiliser `h1` comme fonction de hachage et les mots de la langue française comme clé.
5. Utiliser cette fonction bilan pour vérifier s'il y a eu des collisions, trouver le nombre maximum de mots qui seraient dans la même alvéole, le nombre moyen d'occupation de chaque alvéole, ainsi que l'écart-type. On rappelle que l'écart-type des nombres x_1, x_2, …, x_n est donné par :
$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - m)^2} \quad \text{où} \quad m = \frac{1}{n}\sum_{i=1}^{n} x_i$$
6. On rappelle le principe du paradoxe des anniversaires : si on a un groupe de p personnes, la probabilité qu'au moins deux d'entre-elles aient le même jour d'anniversaire (en négligeant le cas du 29 février) vaut [2] :
$$1 - \prod_{i=0}^{p-1}\frac{365 - i}{365}$$
En particulier, pour 23 personnes, il y a une chance sur deux pour que deux d'entre elles soient nées le même jour. Ici, on peut voir les collisions comme des gens qui sont nés le même jour, ainsi si on a k clés différentes et prises aléatoirement (ce qui n'est pas le cas en pratique) et que l'on a une table de hachage avec n alvéoles, alors la probabilité qu'il y ait (au moins) une collision est :
$$p(n,k) = 1 - \prod_{i=0}^{k-1}\frac{n - i}{n}$$
Que vaut $p(n,k)$ lorsque $n = 10^6$ (correspondant à la fonction de hachage `h1`) et avec `k=len(Liste)`) ? Représenter sur un graphique `p(n,len(Liste))` en fonction de n. Quel n faut-il prendre pour que la probabilité qu'il y ait une collision soit inférieure à $\frac{1}{2}$. Est-ce réaliste de prendre une table de hachage de cette taille ?

2. En effet, calculons la probabilité qu'elles soient tous nées à des jours différents. Alors, chaque personne a 365 possibilités, ainsi le groupe a 365^p possibilités. Si on cherche à compter les possibilités où les gens naissent à des jours différents, alors la première a 365 possibilités, la deuxième 364, la troisième 363 etc.
Au final cela fera $\prod_{i=0}^{p-1}(365-i)$. Comme ici, ce sont des probabilités uniformes $\prod_{i=0}^{p-1}\frac{(365-i)}{365^i}$ représente la probabilité que les personnes soient tous nées des jours différents.

Exercice 7 (**Charité bien ordonnée commence par trier par soi-même.**).

Dans l'exercice 2, on a utilisé la commande `"".join(sorted(mot))` pour trier les lettres de `mot` en boîte noire.

Programmer une fonction qui étant donnée une chaîne de caractères la trie également par ordre alphabétique.

On pourra adapter le tri Fusion ou le tri rapide déjà vu, et placer la lettre `x` avant la lettre `y` si `ord(x)` est strictement plus petit que `ord(y)`.

Exercice 8 (**Et si on hachait autre chose ?**).

1. Dans l'exercice 2, on a une fonction de hachage `h1` qui renvoie un nombre positif pour une chaîne de caractère constitué de lettres minuscules et sans accent. Adaptez `h1` en une fonction `h2` qui puisse contenir les majuscules, les accents et toutes sortes de caractères usuelles.
2. Créer une fonction de hachage `h3` qui hache les nombres et les chaînes de caractères, pour les chaînes de caractères, il suffit de reprendre ce que fait `h2`, pour les entiers relatifs, il suffit de renvoyer le résultat modulo n.

   • Pour un flottant x, on pourra les multiplier par une puissance de 10 de façon à avoir un entier $p = 10^m x$ et renvoyer le résultat de p modulo n.

   • Pour les tuples constitué de nombres ou de chaînes de caractères, on pourra renvoyer la somme des hachage de chacun des éléments du tuple.

   • Mais attention, un tuple peut être constitué lui même de tuples, qui sont eux-même des tuples etc.
   Il faudra donc procéder par récursivité dans ce cas.
   Si `x` est une variable, on pourra utiliser `isinstance(x,tuple)` qui renvoie `True` si `x` est un tuple. (Idem avec `int, float, dict, str` etc.)
3. Compter le nombre de multiplications nécessaires pour la fonction `f(mot)`.
4. Récrire la fonction `h1` qui va d'abord calculer `f(mot)` sans utiliser la fonction `f` mais en utilisant l'algorithme de Horner dans lequel on calcule directement modulo n. Puis compter le nombre de multiplications nécessaires.