TP 4 : Dictionnaires, algorithmes dichotomiques

1 La structure de dictionnaire

Un dictionnaire est un objet permettant de stocker des couples (clé, valeur), chaque clé ne pouvant être présente que dans un seul couple. Un dictionnaire au sens usuel en est un exemple, où les clés sont les mots de la langue et les valeurs les définitions de ces mots. Un autre exemple serait celui d'un annuaire : à partir d'un couple (nom, prénom) d'une personne (la clé), on souhaite retrouver son numéro de téléphone (la valeur).

Syntaxe. Un dictionnaire s'écrit entre accolades, les couples (clé, valeur) indiqués sous la forme `clé:valeur`, séparés par des virgules. Par exemple :

```
>>> D={'a': 4, 0:5, (1,2):"coucou"}
```

Comme on le voit, il n'y a aucune contrainte d'homogénéité sur les clés ou les valeurs.

Opérations sur un dictionnaire. Le plus souvent, on partira d'un dictionnaire vide pour lui rajouter les éléments un à un. Un dictionnaire vide s'obtient via `dict()` ou simplement `{}` :

```
>>> dict(), {}
({}, {})
```

L'accès à un élément `e` à partir de sa clé `k` dans un dictionnaire `D` se fait via `D[k]`, par exemple avec le dictionnaire précédent :

```
>>> D['a'], D[(1,2)]
(4, 'coucou')
```

Si la clé `k` n'est pas présente, l'opération `D[k]` produit une erreur.

```
>>> D['b']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'b'
```

Tester si une clé `k` est présente se fait via `k in D` (ou `k in D.keys()`) :

```
>>> 'b' in D, 'a' in D
(False, True)
```

L'ajout d'un nouveau couple `(k,e)` ou la modification de l'élément `e` associé à une clé `k` se fait de la même manière que l'accès, via `D[k]=e`.

```
>>> D['b']=5 ; D[(1,2)]="coucoubis"
>>> D
{'a': 4, 0: 5, (1, 2): 'coucoubis', 'b': 5}
```

Avertissement : on ne peut pas utiliser un objet mutable (comme une liste, ou un dictionnaire, par exemple) comme clé d'un dictionnaire. Il faut se restreindre à des objets immuables (comme les entiers, les flottants, les chaînes de caractères, les tuples de tels éléments, etc...)

Quelques méthodes et fonctions sur les dictionnaires. Voici quelques méthodes, sans exhaustivité, permises sur les dictionnaires. Dans la suite, `D` désigne un dictionnaire.

méthode	description
`len(D)`	renvoie le nombre de couples (clé, élément) stockés dans `D`.
`D.pop(k)`	supprime le couple de clé `k` du dictionnaire, si un tel couple est présent.
`D.keys()`	renvoie les clés présentes dans le dictionnaire. L'objet renvoyé est un itérable, on peut donc utiliser une syntaxe du type `for k in D.keys(): ...`. Écrire simplement `for k in D` est équivalent.
`D.copy()`	renvoie une copie du dictionnaire.

Complexité ? L'implémentation concrète des dictionnaires est au programme de deuxième année. Vous pouvez considérer que les opérations de création, accès, modification et suppression sont toutes en $O(1)$, sous réserve que les clés utilisées soient de tailles bornées.

2 Exercices sur les dictionnaires

Exercice 1. Écrire une fonction `occurences(L)` prenant en entrée une liste et renvoyant un dictionnaire dont les clés sont les éléments de `L` et les valeurs le nombre de fois où chaque élément apparaît dans `L`. Par exemple :

```
>>> occurences([1, 2, 8, 1, 5, 6, 4, 1, 0, 2, 1, 6, 0])
{1: 4, 2: 2, 8: 1, 5: 1, 6: 2, 4: 1, 0: 2}
```

Exercice 2. En utilisant la fonction précédente :

- Écrire une fonction `sans_doublon(L)` prenant en entrée une liste et renvoyant une liste contenant les mêmes éléments, mais une seule fois.

```
>>> sans_doublons([1, 2, 8, 1, 5, 6, 4, 1, 0, 2, 1, 6, 0])
[1, 2, 8, 5, 6, 4, 0]
```

- Écrire une fonction `doublons(L)` prenant en entrée une liste et renvoyant une liste, contenant tous les éléments de `L` qui apparaîssent au moins deux fois (l'ordre est à votre convenance). Les doublons initiaux apparaîtront une seule fois dans le résultat.

```
>>> doublons([1, 2, 8, 1, 5, 6, 4, 1, 0, 2, 1, 6, 0])
[1, 2, 6, 0]
```

- Écrire une fonction `majoritaire(L)` renvoyant l'élément d'une liste (supposée non vide) qui apparaît le plus souvent (un des éléments en cas d'égalité).

```
>>> majoritaire([1, 2, 8, 1, 5, 6, 4, 1, 0, 2, 1, 6, 0])
1
```

Exercice 3. *Retour sur la suite de Syracuse. Cet exercice est difficile, passez-le dans un premier temps pour y revenir en fin de TP.* On rappelle que la suite de Syracuse est définie comme suit :

$$u_0 \in \mathbb{N}^* \qquad \text{et} \qquad \text{pour } n \geq 1, \quad u_n = \begin{cases} 3u_{n-1}+1 & \text{si } u_{n-1} \text{ impair} \\ u_{n-1}/2 & \text{sinon.} \end{cases}$$

Le *temps de vol* pour $u_0 \in \mathbb{N}^*$ est défini comme le plus petit $n \geq 0$ tel que $u_n = 1$.

1. Écrire une fonction `temps_vol(u0)` calculant le temps de vol de l'entier `u0` supposé strictement positif.

```
>>> temps_vol(15)
17
```

2. En déduire une fonction `max_tps_vol(M)` renvoyant un couple (u_0, t) où u_0 est l'entier de $[\![1, M]\!]$ ayant le plus grand temps de vol, et t le temps de vol associé (en cas d'égalité, on renverra l'un des u_0 atteignant ce maximum).

```
>>> max_tps_vol(100)
(97, 118)
>>> max_tps_vol(10**6) #une quinzaine de secondes sur mon ordinateur personnel.
(837799, 524)
```

3. Les appels précédents font trop de fois les mêmes calculs : par exemple, le temps de vol de l'entier 15 est 17, et les entiers calculés sont les suivants :

$$15 \to 46 \to 23 \to 70 \to 35 \to 106 \to 53 \to 160 \to 80 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

On pourrait en déduire du même coup que le temps de vol de 46 est 16, celui de 23 est 15, etc... De plus, on aurait pu arrêter le calcul dès qu'un entier dont le temps de vol est connu est atteint. Si le temps de vol de 10 avait déja été calculé auparavant, on aurait pu s'arrêter là. Écrire une fonction `tous_tps_vol(M)` prenant en entrée un entier strictement positif, et stockant dans un dictionnaire les temps de vol de tous les entiers de $[\![1, M]\!]$, ainsi que des entiers atteints lorsque u_0 est un des entiers de $[\![1, M]\!]$. On suivra le schéma suivant (sans appeler `temps_vol` !) :

```
def tous_tps_vol(M):
    D={} #dictionnaire vide
    D[1]=0 #temps de vol de 1 : 0
    for u in range(2,M+1):
        [ ... calcul des termes de la suite depuis u, jusqu'à tomber sur un terme dont le temps
          de vol est connu. On stockera les termes calculés dans une liste, puis on stockera dans D
          les temps de vol de tous les entiers ainsi découverts ...]
    return D
```

Par exemple :

```
>>> tous_tps_vol(10)
{1: 0, 2: 1, 4: 2, 8: 3, 16: 4, 5: 5, 10: 6, 3: 7, 6: 8, 20: 7, 40: 8, 13: 9, 26: 10, 52: 11,
17: 12, 34: 13, 11: 14, 22: 15, 7: 16, 14: 17, 28: 18, 9: 19}
```

4. En déduire une fonction `max_tps_vol2(M)` faisant la même chose qu'à la question 2, en utilisant la fonction précédente. Vérifiez que le calcul est plus rapide ! Sur mon ordinateur personnel, l'appel `max_tps_vol2(10**6)` prend moins de deux secondes.

3 Algorithmes dichotomiques

On a vu en cours un algorithme permettant de tester si un élément était présent dans une liste triée, voici pour rappel le code :

```
def dicho(L,x):
    """ L triée, teste si x est présent dans L """
    g,d = 0, len(L)
    #Invariant : x n'est ni dans L[:g] ni dans L[d:]
    while d-g>0:
        m=(g+d)//2
        if L[m]==x:
            return True
        elif L[m]<x:
            g=m+1
        else:
            d=m
    return False
```

Dans cette section, on propose des variantes. Réfléchissez bien à l'invariant de boucle que vous voulez maintenir !

Exercice 4. *Recherche d'un zéro d'une fonction.* Soit $[a,b]$ un intervalle ($a < b$) et $f : [a,b] \to \mathbb{R}$ une fonction continue, telle que $f(a)f(b) \leq 0$ (ainsi f change de signe entre les deux extrémités de l'intervalle). Le théorème des valeurs intermédiaires assure que f s'annule au moins une fois sur l'intervalle. Écrire une fonction `zero(f,a,b,eps)` prenant en entrée une telle fonction, les extrémités de l'intervalle, et un petit flottant ε, et renvoyant une approximation à ε près d'un zéro de f. Par exemple :

```
>>> from math import *
>>> zero(sin, 2, 4, 10**-5) #sin(2)>0, sin(4)<0. Il n'y a ici qu'un seul zéro sur l'intervalle : pi
3.1415939331054688
>>> zero(lambda x:x**2-2, 0, 5, 10**-8) #lambda x: expression permet de définir une fonction anonyme
1.4142135623842478
```

Remarque : l'algorithme que vous allez écrire existe déja, dans le module ***scipy.optimize****, sous le nom* ***bisect****.*

Exercice 5. *Variante de la recherche dichotomique dans une liste.*

- On se donne une liste `L` supposée strictement croissante, et un élément x vérifiant `L[0]` $\leq x <$ `L[len(L)-1]`. Il existe un unique indice i vérifiant `L[i]` $\leq x <$ `L[i+1]`. Écrire une fonction `indice_inf(L,x)` renvoyant cet indice en temps logarithmique en la taille de la liste.

```
>>> indice_inf([0,2,4,6,8,10], 3)
1
>>> indice_inf([0,2,4,6,8,10], 4)
2
```

- Soit `f` une fonction de $\mathbb{N} \to \{0,1\}$, croissante et surjective. Écrire une fonction `point_changement(f)` donnant l'unique entier x tel que $f(x) = 0$ et $f(x+1) = 1$, avec une complexité de $O(\log x)$ appels à `f`. Par exemple :

```
def g(x):
    if x>1648464866186:
        return 1
    else:
        return 0
```

L'appel `point_changement(g)` renvoie 1648464866186 (instantanément, ce qu'une recherche linéaire ne permet absolument pas).

Exercice 6. *Recherche de pic dans une liste.* On dit qu'un indice i d'une liste `L` est un pic si l'élément `L[i]` est supérieur ou égal à ses deux voisins (ou son unique voisin s'il est sur le bord). Par exemple, dans la liste `[10, 51, 32, 14, 78, 40, 82]`, les pics sont 1, 4 et 6.

1. Écrire une fonction `test_pic(L)` prenant en entrée une liste non vide et un indice de la liste, et testant si celui-ci est un pic (votre fonction renverra naturellement un booléen). *Attention si $i = 0$ ou $i =$ `len(L)` $- 1$, à ne pas tenter d'accéder à des éléments qui n'existent pas, cela produirait une erreur !*
2. En déduire une fonction `pics(L)` prenant en entrée une liste et renvoyant la liste de ses pics.

```
>>> pics([1, 2, 8, 1, 5, 6, 4, 1, 0, 2, 1, 6, 7])
[2, 5, 9, 12]
```

On veut maintenant écrire une fonction recherchant *un seul pic* dans une liste, mais en temps logarithmique en la taille de la liste. Remarquons les faits suivants :

- Déja, une liste non vide possède toujours au moins un pic : l'indice du maximum.
- Si i est un indice de la liste qui n'est pas un pic, alors au moins l'une des deux conditions suivantes est vérifiée : $i \geq 1$ et `L[i]` $<$ `L[i-1]`, ou $i \leq$ `len(L)` $- 2$ et `L[i]` $<$ `L[i+1]`.
  - Si $i \geq 1$ et `L[i]` $<$ `L[i-1]`, alors un pic p de la liste `L[:i]` (liste constituée des éléments de `L` d'indice entre 0 et $i-1$) est nécessairement un pic de `L` : c'est évidemment le cas si $p < i-1$, et si $p = i-1$ c'est le cas également puisque `L[i]` $<$ `L[i-1]`. On peut donc chercher un pic à un indice strictement inférieur à i dans ce cas.
  - Symétriquement, si $i \leq$ `len(L)` $- 2$ et `L[i]` $<$ `L[i+1]`, on peut chercher un pic d'indice supérieur ou égal à $i+1$.

3. En vous inspirant de la discussion précédente, écrire une fonction `pic_dicho(L)` prenant en entrée une liste non vide et permettant de déterminer l'indice d'un pic de la liste `L` avec une complexité $O(\log n)$ où n est la taille de la liste. On ne fera pas d'extraction de portions de liste (coûteux en complexité). *Attention à ne pas faire de dépassement d'indice !*

```
>>> pic_dicho([1, 2, 8, 1, 5, 6, 4, 1, 0, 2, 1, 6, 7])
2
>>> pic_dicho(list(range(10)))
9
>>> pic_dicho([10])
0
```