TP5 - Dichotomie

Comme d'habitude, on lance spyder (dans Mathématiques→WinPython), on crée un répertoire TP5 dans le répertoire InfoPCSI de votre session, et on y sauvegarde le script de l'éditeur de code (fenêtre de gauche) sous le nom `tp5.py`.

L'objet de ce TP va d'abord être de reprendre une fonction abordée au cours du dernier TP (`indiceDe`) où on cherchait où une valeur pouvait être présente dans une liste donnée (à quel indice donc). Sans information supplémentaire, on n'a guère d'autre moyen que de faire une recherche dite séquentielle, où on parcourt une fois tous les termes de notre liste, en s'arrêtant dès que la valeur est trouvée (mais en devant aussi lire tous les termes de notre liste une fois dans le cas où la valeur recherchée ne figure pas).

Comme suggéré dans le TP sur les boucles imbriquées, le contexte est tout à fait différent si notre liste a le bon goût d'être triée, car si on commence par tester la valeur médiane de notre liste, la recherche se cantonnera dès la deuxième étape à la moitié des termes de celle-ci.

1 Etude d'un exemple

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	3	6	7	7	10	10	10	11	12	14	16	19	19	19	19	20

Nous tâcherons de répondre à la même question que celle à laquelle répond `indiceDe` du TP précédent, à savoir de renvoyer le premier indice où figure l'objet recherché, ou bien `None` en cas d'absence. L'idée sera au gré de quelques tests de réduire la zone de recherche (dans l'idéal, en diviser la longueur par deux à chaque test), jusqu'à ce que celle-ci ne comporte plus qu'un ou zéro terme.

1.1 Exemples naïfs

Si on cherche l'entier 10 par exemple, on commence par tester la valeur médiane, à savoir `L[8]` qui vaut 11. On sait qu'alors si la valeur est présente dans la liste, elle le sera entre les indices 0 et 7. On regarde alors `L[3]` et on se cantonne à la recherche entre les indices 4 et 7. Puis on compare à `L[5]` qui vaut 10 et on a trouvé la valeur recherchée.

En toute rigueur, ce n'est pas complètement terminé, car on ne sait pas si `5` est le premier indice où apparaît la valeur 10. (Vu ce qui précède, ce pourrait aussi être `4`.)

Lorsqu'on tombe sur la valeur recherchée, la stratégie qui consiste à tester toutes les valeurs précédentes de la liste jusqu'à trouver une valeur qui diffère de celle que l'on souhaite n'est pas la bonne, car elle pourrait conduire à tester beaucoup trop de valeurs et finalement perdre l'avantage que permet la dichotomie. (Imaginez par exemple une liste où tous les termes sont égaux...)

Question 1. Décrire les étapes qui permettraient de trouver 15, 19 en prenant exemple sur ce qui précède.

1.2 Automatisation de la recherche

Il n'est pas si facile de coder une dichotomie : déjà, quand on compare une valeur de `L` telle que `L[i]` à la valeur recherchée, mettons `v`, doit-on réaliser les tests `L[i]<v`, `L[i]>v`, `L[i]<=v`, `L[i]>=v`, `L[i]==v` ? (Tous certainement pas bien sûr, mais une seule de ces comparaisons ? plusieurs ?)

Le principe sera bien sûr que la zone de recherche englobe toujours l'indice solution de notre problème s'il existe, mais aussi de s'assurer que celui-ci sera, en toutes circonstances, bel et bien trouvé (ou que la valeur `None` puisse être renvoyée), pourvu de laisser suffisamment de temps à notre algorithme pour s'exécuter : on touche là à la notion de preuve de correction et de terminaison d'un algorithme. On va voir ici en quoi en se donnant la peine de prouver que notre algorithme fonctionne, les spécifications que l'on va imposer vont, d'une certaine manière, nous guider vers celui-ci.

Pour spécifier la zone de recherche dans `L`, on va délimiter celle-ci par deux indices `a` et `b` ainsi celle-ci sera formée des éléments de `L` d'indice `i` compris au sens large entre `a` et `b`.

D'où la première condition qu'on cherchera à satisfaire, tout au long de notre recherche : « si `v` figure dans `L` et que `k` est le premier indice où `v` est présent, alors `a<=k<=b` ».

On initialisera `a` et `b` aux valeurs `0` et `len(L)-1` (de sorte que les indices de `L`, on s'en souvient, sont les indices `i` tels que `a<=i<=b` et ainsi la zone de recherche initiale comprend bien sûr l'intégralité de `L`)

Question 2. On suppose vérifiée, à un moment donné, la propriété : « si `v` figure dans `L` et que `k` est le premier indice où `v` est présent, alors `a<=k<=b` » et on pose `c=(a+b)//2` (la valeur moyenne `(a+b)/2` n'est pas forcément un entier, c'est pourquoi on prend le quotient de la division euclidienne de `a+b` par `2`)

Dans chacun des quatre cas suivants, indiquer par quelles valeurs remplacer `a` et/ou `b` pour que la propriété rappelée ci-dessus soit encore satisfaite (attention : il est un cas où il n'existe pas réellement de réponse satisfaisante !)

1. `L[c] <= v`
2. `L[c] >= v`
3. `L[c] < v`
4. `L[c] > v`

Question 3. Etant donnés `a<b` deux entiers, et en posant `c=(a+b)//2` (le quotient donc de la division euclidienne de `a+b` par 2), déterminer à la main la valeur de `c` pour les couples `(a,b)` suivants :

`(1, 6), (2, 6), (3, 5), (3, 4), (4, 5)`

Question 4. On a sans surprise toujours, en notant encore `c=(a+b)//2`, les inégalités `a<=c<=b`.

De plus, en supposant `a<b`, l'une de ces deux inégalités est toujours stricte. Laquelle, et pourquoi ?

Question 5. On va reprendre à la main l'exemple présenté au début du paragraphe pour la recherche du premier indice `c` où figurent les valeurs `14,19` et `21`.

On partira bien sûr de `a=0` et `b=16`, on posera à chaque étape `c=(a+b)//2`, et à chaque étape on se permettra un et un seul test, à savoir : `L[c] < v`. Selon que ce test est ou n'est pas satisfait, on modifiera `a` et/ou `b`, et on recommence jusqu'à ce que `a` et `b` soient égaux. On pourra compléter un tableau de la forme (le nombre de lignes n'est ici pas suffisant) :

`v = 14` :

a	b	c	L[c] < v
0	16	8	oui
9	16	…	…

2 Implémentation

Introduisez dans votre fichier `tp5.py` le code suivant :

```
from random import randrange

def aléatoireTriée(n):
    L = []
    v = randrange(10)
    L.append(v)
    for i in range(n):
        v = v + randrange(3)
        L.append(v)
    return L
```

Question 6. Que réalise l'instruction `aléatoireTriée(100)` ?

Question 7. Compléter la fonction suivante, qui réalise la recherche dichotomique de `v` dans une liste supposée triée dans le sens croissant `L`, et qui renvoie le premier indice `i` où figure `v` dans `L` en cas de présence, et `None` sinon (petit rappel : `None` est la valeur de retour par défaut, donc il n'est pas nécessaire de terminer par un `return None`) :

```
def rechDichotomique(L, v):
    a, b = 0, len(L)-1
    while ...:
        c = (a+b) // 2
        ...
    if v == L[...]:
        return ...
```

3 Vérification empirique de la complexité

De la dichotomie, on s'attend à ce que, en doublant le nombre de termes d'une liste, la recherche dichotomique nécessite un tour de boucle supplémentaire (c-à-d presque rien…). Le temps d'exécution de la recherche dichotomique sur une liste de quelques milliers de termes étant, on s'en doute, très court, le protocole pour mesurer le temps d'exécution moyen sera le suivant :

- On crée une liste triée `L` de n termes une fois pour toutes à l'aide de la fonction `aléatoireTriée`
- On démarre alors le chronomètre, et on effectue un grand nombre de fois (mettons 10000 fois…) l'opération suivante :

  On tire un nombre aléatoire `v`, mettons entre 0 et $2\,n$ et on effectue la recherche de `v` dans `L`.
- On arrête le chronomètre.

En négligeant le temps que coûte la génération d'un nombre (pseudo-)aléatoire, on aura une bonne approximation du temps moyen à une recherche dichotomique parmi n termes.

Question 8. Ecrire une fonction d'entête `def tempsMoyen(n):` qui renvoie le temps moyen d'une recherche dichomotique sur une liste de n termes. On prendra les valeurs de n suivantes : 1000, 2000, 4000, 16000.

4 Quelques variantes

Voici deux autres problèmes qui sont du ressort d'une dichotomie, toujours en supposant `L` une liste triée dans le sens croissant :

- Etant donné `v`, déterminer l'indice `a` à partir duquel tous les termes de `L` seront au moins égaux à `v`. Si tous les termes de `L` sont strictement inférieurs à `v`, on renverra `len(L)`. (On note que `L[a:]` est alors la liste formée de tous les termes de `L` valant au moins `v`, et cette syntaxe prend encore du sens si `a=len(L)`, mais la liste alors obtenue est la liste vide)
- Etant donné `v`, déterminer l'entier `a` compris entre `0` et `len(L)` tel que `L[:a]` soit la liste extraite de `L` formée de tous les éléments au plus égaux à `v`. On rappelle que `L[:a]` sera la liste vide si `a=0`, et la liste formée de `L[0],...,L[a-1]` sinon.

Question 9. Pour le premier des deux problèmes précédents, en donner une réponse en modifiant judicieusement la fonction `rechDichotomique` de la question 7.

Question 10. Ecrire une fonction qui répond au second des deux problèmes.

5 Annexe : quelques éléments de syntaxe python

1. **Boucles `for` et `range`** : `range` est un constructeur qui nous sert à définir un ensemble d'entiers sur lesquels itérer, et qu'on utilise essentiellement pour créer des boucles `for`. Avec un seul argument `n`, `range(n)` va permettre d'itérer sur les valeurs de `0` à `n-1`. Avec deux arguments `a` et `b`, `range(a, b)` va itérer sur les entiers de `a` à `b-1`, et enfin on peut rajouter un troisième argument, lequel est le pas (par défaut de 1, mais on peut aller de 2 en 2, à l'envers avec un pas de -1). Ainsi par exemple, `range(10,2,-2)` va itérer sur les entiers de 10 à 2 de -2 en -2, l'indice final n'étant pas compris (il ne l'est jamais) c'est-à-dire sur les entiers 10, 8, 6, 4.

2. La fonction `randrange` du module `random` permet de générer des nombres pseudoaléatoires : `randrange(n)` fournit un nombre entre `0` et `n-1`, `randrange(a, b)` un nombre `k` tel que `a<=k<b` (une erreur est générée si aucun entier ne peut satisfaire ces inégalités)…

3. **Listes** : étant donnée une liste `L`, `len(L)` donne le nombre de termes de `L`. Si `n` est ce nombre, alors les indices des termes de `L` sont numérotés de `0` à `len(L)-1` et on accède, en lecture et en écriture, au terme d'indice `i` par la syntaxe `L[i]`.

   Deux méthodes à connaître : `append` et `pop` : l'instruction `L.append(val)` rajoute à la fin de `L` le terme `val`, tandis que `L.pop()` extrait le dernier terme de `L` (le retire) et en renvoie la valeur

4. `perf_counter` : on rappelle un exemple d'utilisation :

```
from time import perf_counter
start = perf_counter()
unTraitementLong()
stop = perf_counter()
tempsEcoulé = stop - start # contient le temps écoulé dans unTraitementLong() en secondes
```