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 d'un TP ancien (`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 seulterme.

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 chercher la présence éventuelle de 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 aussi 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`

Réponse.

1. On pourrait être tenté de remplacer `a` par `c` et de laisser `b` inchangé, mais ce serait une erreur, car dans le cas où `L[c]==v` alors rien ne prouve que le premier indice où `v` apparaît soit `c` ! (En toute rigueur, il pourrait s'agir de tout indice compris au sens large entre `a` et `c`.)

   Bref, ce test ne nous aide guère à réduire notre zone de recherche, pour le problème posé en tout cas de déterminer le premier indice où apparaît `v`.

2. Là, une chose est certaine, le premier indice où apparaît `v` doit être compris, s'il existe, au sens large, entre les indices `a` et `c`. Ce qui indique qu'on pourrait remplacer `b` par `c`.

3. Ici encore, il ne fait aucun doute que l'indice recherché, s'il existe, figure au sens large entre `c+1` et `b` donc on pourrait remplacer dans ce cas `a` par `c+1`.

4. Ici, on pourrait remplacer `b` par `c-1`.

On notera que le test `L[c] >= v` et sa négation a priori permettent l'un comme l'autre de réduire notre zone de recherche. Le test `L[c] > v` le permet aussi, mais pas sa négation, ce qui nous invite plutôt à utiliser le test précédent (ou sa négation, avec un branchement conditionnel `if ...: else:` ce qui permet de ne réaliser qu'une seule comparaison.)

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)`

Réponse. `3, 4, 4, 3` et `4`.

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 ?

Réponse. On a toujours `a<=c<b` car `a<(a+b)/2<b` et `a<=(a+b)//2<=(a+b)/2`.

Question 5. On va reprendre à la main l'exemple présenté au début du paragraphe pour la recherche du premier indice `c` (s'il existe) 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)` ?

Réponse. elle crée une liste triée de 101 valeurs (il y a un piège !) où la différence entre deux termes consécutifs vaut 0,1 ou 2. (Le premier terme lui prend une valeur entre 0 et 9)

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 ...
```

Réponse.

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

Question 8. A l'aide des questions précédentes, expliquer pourquoi, partant d'une liste non vide `L` et un objet `v`, alors la boucle conditionnelle de la fonction précédente finit bien par s'achever (preuve de terminaison), et pourquoi, si `v` figure dans `L`, alors notre fonction renvoie bien le premier indice `i` où figure `v`, et qu'elle renvoie `None` dans le cas contraire (preuve de correction).

Réponse. On a introduit la propriété suivante : « si `v` figure dans `L`, alors le premier indice `i` tel que `L[i]=v` vérifie `a<=i<=b` »

On justifie que cette propriété (on lui donnera le nom d'invariant de boucle) est satisfaite à l'entrée dans la boucle, ainsi qu'au début et à la fin de chaque itération (et de ce fait aussi lorsque la boucle termine)

Initialement, `a=0` et `b=len(L)-1` et la propriété énoncée est manifestement satisfaite.

Si la propriété est satisfaite au début d'une itération, alors si `L[c] < v`, le premier indice `i` s'il existe vaut au moins `c + 1` et donc la propriété est encore satisfaite en remplaçant `a` par `c + 1`, et dans le cas contraire (si `L[c] >= v`), ce premier indice `i` s'il existe est au plus égal à `b` et la propriété est bien encore satisfaite en remplaçant `b` par `c`. Bref, dans tous les cas, la propriété est encore satisfaite à la fin de l'itération.

De ce fait, lorsque la boucle termine, on a `a = b`, et le premier indice où `v` figure dans `L`, s'il existe, est l'indice `a`, donc on n'a plus qu'à tester si `v` figure bel et bien dans `L` à l'indice `a`, ou non (auquel cas, `v` ne figure pas dans `L`, tout court)

Pour prouver la terminaison de l'algorithme, on a vu que la valeur `b-a` est celle d'un entier positif, et qu'elle décroit à chaque itération, et qu'il ne saurait donc y avoir davantage que `len(L)-1` itérations, donc on est certain, au moins d'un point de vue théorique, que notre boucle ne pourra être une boucle infinie.

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

Réponse. Par exemple :

```
from time import perf_counter

def tempsMoyen(n):
    L = aléatoireTriée(n)
    start = perf_counter()
    for i in range(10000):
        rechDicho(L, randrange(2*n))
    stop = perf_counter()
    return (stop-start)/10000
```

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

Réponse. On reprend la fonction de la question 7. Lorsque la boucle conditionnelle s'interrompt, c'est que la zone de recherche ne comprend plus qu'un seul terme de `L`, celui d'indice `a=b`. Ce qui est certain, c'est que les termes de `L` qui suivent le terme d'indice `a` (si celui d'indice `a` n'est pas le dernier) sont au moins égaux à `v`, et que ceux qui le précèdent (s'il y en a) sont strictement inférieurs à `v`. Donc l'indice recherché vaut ou bien `a` (si `L[a]<=v`) ou bien `a+1` (si `L[a]>v`).

Il n'y a donc que la toute fin de la fonction à reprendre :

```
def rechDichotomique2(L, v):
```

```
    a, b = 0, len(L)-1
    while b > a:
        c = (a+b) // 2
        if L[c] < v:
            a = c + 1
        else:
            b = c
    if L[a] >= v:
        return a
    else:
        return a + 1
```

Voici une version proposée par plusieurs d'entre vous :

```
def rechDichotomie_inf(L, v):
    a, b = 0, len(L)
    while a < b:
        c = (a + b) // 2
        if L[c] < v:
            a = c + 1
        else:
            b = c
    return a
```

Elle est excellente, mais il reste à se convaincre qu'elle répond bien au problème posé. Faisons une preuve de correction de cet algorithme : on énonce la propriété suivante, « les termes de `L` d'indices strictement inférieurs à `a` (s'il y en a) sont strictement inférieurs à `v` et les termes de `L` d'indices au moins égaux à `b` sont au moins égaux à `v` » qu'on appellera bientôt invariant de boucle, et on justifie que cette propriété est toujours satisfaite (à l'entrée dans la boucle, au début et à la fin de chaque itération, et donc encore lorsque la boucle s'interrompt) :

à l'entrée dans la boucle, `a` vaut 0 donc il n'y a aucun terme dans `L` d'indice strictement inférieur à `a`, ni de terme d'indice au moins égal à `b` qui vaut `len(L)`, ainsi la propriété est satisfaite à l'entrée dans la boucle.

Si la propriété est satisfaite au début d'une itération, en remplaçant `a` par `c + 1` si `L[c] < v`, alors il est manifestement vrai que les termes de `L` d'indices strictement inférieurs à `a` sont strictement inférieurs à `v`, et dans le cas où on remplace `b` par `c`, il est tout aussi évident que les termes d'indices au moins égaux à `b` valent au moins `v`.

Autrement dit, lorsque la boucle s'interrompt, c'est qu'on a `a` et `b` égaux, et que `a` est bien la réponse attendue.

Pour montrer que notre algorithme termine (et qu'on ne se retrouve donc pas avec une boucle infinie), on s'appuie sur les mêmes arguments que pour la première fonction de dichotomie : la valeur de `b-a` décroit strictement à chaque itération, et reste positive, donc on ne saurait avoir davantage que `len(L)` itérations (en pratique, il y en a beaucoup moins).

Question 11. Ecrire une fonction qui répond au second des deux problèmes.

Réponse. On peut reprendre les arguments des premières questions, en cherchant le premier indice `k` tel que `L[k]>v` si un tel indice existe. Si `k` est celui-ci, et si `a<=k<=b` alors en posant `c=(a+b)//2`, en comparant `L[c]` à `v`, on voit que si `L[c]<=v`, alors `c<k<=b` et si `L[c]>v`, alors `a<=k<=c`.

Lorsque notre dichotomie s'interrompt, en suivant le principe ci-dessus, on obtient `a=b` et tous les termes de `L` à droite (strictement) de `a` sont strictement supérieurs à `v`, et tous ceux à gauche (strictement) sont inférieurs ou égaux à `c`.

Il n'y a plus qu'à regarder la valeur de `L[a]` pour conclure.

```
def rechDicho2(L, v):
    a, b = 0, len(L)-1
    while b > a:
        c = (a+b) // 2
        if L[c] <= v:
            a = c+1
        else:
            b = c
    if L[a] <= v:
        return a + 1
    else:
        return a
```

Ici aussi, plusieurs d'entre-vous ont su donner une réponse plus élégante que je reproduis ici :

```
def rechDichotomie_sup(L, v):
    a, b = 0, len(L)
    while a < b:
        c = (a + b) // 2
        if L[c] <= v:
            a = c + 1
        else:
            b = c
    return a
```

Je vous laisse énoncer un invariant de boucle grâce auquel on montre la correction de cet algorithme.

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 avec un pas de 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
   ```

6 Travail à la maison : calcul de puissances, exponentiation rapide

Les réponses aux questions 10 et 11 (ce qui n'est pas si facile, je le reconnais, donc j'accepte aussi les questions) et les deux questions qui suivent :

Parmi les algorithmes dits dichotomiques figure aussi l'algorithme d'exponentiation rapide, dont l'objet est de calculer efficacement une puissance x^n où n est un grand entier. L'algorithme naïf consiste à calculer au minimum $n-1$ multiplications (en pratique, plus probablement n, car on partira de 1 qu'on multipliera n fois par x…

Question 12. Ecrire une fonction d'entête `def puissance(x, n)` qui prend en argument un nombre `x` (entier, flotant, complexe, peu importe) et un entier naturel `n` et qui renvoie le nombre x^n. On s'interdira bien sûr l'opérateur d'exponentiation `**`, les fonctions exponentielle et logarithme.

On se rend vite compte que l'algorithme naïf présenté précédemment n'est pas optimal. Imaginons que nous ayons à calculer x^{16}, alors, $x^{16} = (x^8)^2 = ((x^4)^2)^2 = (((x^2)^2)^2)^2$ et on voit que 4 multiplications (ici des élévations au carré) suffisent à calculer x^{16}. Pour x^{19}, on voit qu'il se décompose en $x\,x^2\,(((x^2)^2)^2)^2$ et que 6 multiplications suffisent (ou plutôt 7, en partant de 1). C'est la décomposition de 19 en base 2 qui donne ce produit, car $19 = 1+2+2^4$ (ce qui exprime qu'en base 2, 19 s'écrit 10011)

On rappelle que l'écriture en base b d'un entier s'obtient facilement par une suite de divisions euclidiennes par b. Par exemple, pour retrouver un à un les chiffres de l'entier 123, sa division euclidienne par 10 donne $123 = 10\times 12+3$ qui indique que le chiffre des unités vaut 1. Puis on écrit $12 = 10\times 1+2$ ce qui donne le chiffre suivant 2, des dizaines, de son écriture en base 10. On écrit encore $1 = 10\times 0+1$ et 1 est le dernier chiffre, des centaines, de l'écriture de 123 en base 10.

En base 2, c'est la même chose :

Divisions euclidiennes	Ecriture binaire	p	c
$19 = 2 \times 9 + 1$	????1	x	x
$9 = 2 \times 4 + 1$	???11	x^3	x^2
$4 = 2 \times 2 + 0$	??011	x^3	x^4
$2 = 2 \times 1 + 0$	?0011	x^3	x^8
$1 = 2 \times 0 + 1$	10011	x^{19}	x^{16}

L'algorithme d'exponentiation rapide fonctionne de la manière suivante :

- on initialise une variable `p` à la valeur 1 : elle contiendra à la fin la puissance de x à calculer, et on initialise une variable `c` à la valeur x. A (la fin de) chaque étape, on élève `c` au carré (voir tableau ci-dessus)

- On réalise des divisions euclidiennes répétées, de notre exposant n par 2, où on remplace à chaque étape n par le quotient de notre division euclidienne. Lorsque le reste de la division euclidienne vaut 1, on modifie `p` pour le multiplier par la valeur de `c`

- Bien sûr, notre algorithme s'arrête quand, à force de divisions euclidiennes répétées, on parvient à un quotient nul.

Question 13. Implémenter l'algorithme d'exponentiation rapide dans une fonction d'entête `def expoRapide(x, n)`.

En reprenant l'exemple de la question 8, justifier la correction de l'algorithme précédent.

Bien entendu, vos scripts sont à envoyer sous la forme d'un fichier python par mail à l'adresse `phjondot@gmail.com`