ENSAE TD noté, mardi 27 décembre 2013

Le programme construit au fur et à mesure des questions devra être imprimé à la fin du TD et rendu au chargé de TD. ***Il ne faut pas oublier de mentionner son nom inséré en commentaire au début du programme et de l'ajouter sur chaque page.*** *Les réponses autres que des parties de programme seront insérées sous forme de commentaires. Les squelettes de fonctions proposés ne sont que des suggestions.* ***Il faudra aussi indiquer le numéro des exercices sous forme de commentaires.***

1

La recherche dichotomique est assez simple. On cherche un élément e dans un tableau trié :

1. On compare e à l'élément du milieu.
2. S'ils sont égaux, on s'arrête. Si e est inférieur, on cherche dans la première partie du tableau. On cherche dans la partie supérieure dans l'autre cas.

On part de l'algorithme implémenté ici de façon récursive ou non et disponible ici : http://www.xavierdupre.fr/blog/2013-12-01_nojs.html.

1) Récupérez l'algorithme de la recherche dichotomique et appliquez-le à la liste [0, 2, 4, 6, 8, 100, 1000] et le nombre 100. (1 point)

2) On suppose qu'on dispose de deux listes triées de nombres, une pour les nombres impairs et une autre pour les nombres pairs. On veut écrire une fonction qui recherche un entier dans une de ces deux listes sachant que la parité du nombre indique dans quelle liste chercher et que le nombre cherché s'y trouve.

```
def deux_recherches(element, liste_impaire, liste_paire) :
    # ....
    return position_dans_liste_impaire, position_dans_liste_paire
```

Vous pouvez appliquer votre fonction à l'élement 100 et aux listes [0, 2, 4, 6, 8, 100, 1000], [1,3,5]. Le résultat attendu est (-1,5).

(2 points)

3) On suppose que les deux listes (nombres impairs, nombres pairs) sont de même tailles n. Dans le cas d'une recherche simple, le fait de couper la liste en deux est un avantage, est-ce toujours le cas dans le cas d'une recherche dichotomique ? Justifiez. (1 point)

4) Adaptez l'algorithme (de la question 1) pour retourner -1 lorsque l'élément à chercher n'est pas dans la liste. Il peut être utile de vérifier que vous arrivez bien à retrouver tous les éléments. (3 points)

```
l = [ 0, 2, 4, 6, 8, 100, 1000 ]
for i in l :
    print (i,recherche_dichotomique(i, l))
```

5) On modifie maintenant la fonction (de la question 2) de telle sorte qu'on cherche d'abord dans la liste des nombres impairs et si on ne trouve pas, on cherche dans la liste des nombres pairs. (2 points)

6) On doit chercher 1000 nombres impairs et 1 nombre pair. Quelle est la fonction la plus rapide pour $\ln_2(n) = 30$ ($n = 2^{30} \sim 10^9$)) ? Pourquoi ? (1 point)

Correction

```
#coding: latin-1

######### énoncé 1, exercice 1, recherche dichotomique

## question 1

def recherche_dichotomique( element, liste_triee ):
    """
    premier code: http://www.xavierdupre.fr/blog/2013-12-01_nojs.html
    """
    a = 0
    b = len(liste_triee)-1
    m = (a+b)//2
    while a < b :
        if liste_triee[m] == element :
            return m
        elif liste_triee[m] > element :
            b = m-1
        else :
            a = m+1
        m = (a+b)//2
    return a

l = [ 0, 2, 4, 6, 8, 100, 1000 ]
print (recherche_dichotomique(100, l))  # affiche 5

## question 2

def deux_recherches(element,liste_impair,liste_pair) :
    if element % 2 == 0 :
        return recherche_dichotomique(element, liste_pair), -1
    else :
        return -1, recherche_dichotomique(element, liste_impair)

lp = [ 0, 2, 4, 6, 8, 100, 1000 ]
li = [ 1, 3, 5 ]
print (deux_recherches(100, li, lp))  # affiche (5, -1)

## question 3
"""
                            liste coupée       liste non coupée (2n)
recherche simple                1 + n                 2n
recherche dichotomique          1 + ln n             ln(2n) = 1 + ln(n)

coût équivalent
"""

## question 4

def recherche_dichotomique( element, liste_triee ):
    a = 0
    b = len(liste_triee)-1
    m = (a+b)//2
    while a < b :
        if liste_triee[m] == element :
            return m
        elif liste_triee[m] > element :
```

```
            b = m-1
        else :
            a = m+1
        m = (a+b)//2
    if liste_triee[a] != element :  return -1       # ligne ajoutée
    else : return m                                 # ligne ajoutée

l = [ 0, 2, 4, 6, 8, 100, 1000 ]
for i in l :
    print (i,recherche_dichotomique(i, l))
    # vérifier qu'on retrouve tous les éléments existant
print (recherche_dichotomique(1, l))   # affiche -1

## question  5

def deux_recherches(element,liste_impair,liste_pair) :
    i = recherche_dichotomique(element, liste_impair)
    if i == -1 : return recherche_dichotomique(element, liste_pair)
    else : return i

##question 6

"""
Les logarithmes sont en base 2.

coût fonction question 2 : 1001 ( 1 + ln(n) )      = C1
coût fonction question 5 : 1000 ln(n)  + 2 ln(n)   = C2
C2 - C1 = ln(n) - 1001 > 0

La fonction 5 est plus rapide dans ce cas.
"""
```

Quelques remarques :

1. Question 2 : il était préférable d'appeler la première fonction plutôt que de recopier son code deux fois.
2. Question 4 : il était simple d'ajouter la ligne A à la fonction de la question 1 mais la recherche n'est plus dichotomique. Rechercher un élément dans une liste avec le mot clé `in` revient à passer les éléments de la liste en revue. La fonction fait alors deux recherches : une non-dichotomique et une seconde dichotomique.

   ```
   def recherche_dichotomique( element, liste_triee ):
       if element not in list_triee : return -1 # ligne A
       a = 0
       b = len(liste_triee)-1
       m = (a+b)//2
       while a < b :
           if liste_triee[m] == element :
               return m
           elif liste_triee[m] > element :
               b = m-1
           else :
               a = m+1
           m = (a+b)//2
       return a
   ```

   J'ai enlevé un point.

3. Question 5 : il faut faire attention à ne pas appeler la fonction `recherche_dichotomique` plus de fois que nécessaire. Dans le cas qui suit, on recherche soit une fois dans chaque liste, soit deux fois dans la liste `liste_impair`. J'ai enlevé 0.5 point.

```
def deux_recherches(element,liste_impair,liste_pair) :
    if recherche_dichotomique(element, liste_impair) == -1 :
       return recherche_dichotomique(element, liste_pair)
    else : return recherche_dichotomique(element, liste_impair)
```

4. Question 6 : pour ces questions de coûts, il est préférable d'éviter des affirmations qualitatives du type *un coût qui s'avère fatal*. J'ai enlevé 0.5 point.

2

On s'intéresse à la distance de Levensthein qui est une distance entre deux mots. Pour deux mots $L = (l_1, .., l_m)$ et $K = (k_1, ..., k_n)$, elle se résume par la relation de récurrence suivante :

$$d(i,j) = \min \left\{ \begin{array}{l} d(i-1,j)+1 \\ d(i,j-1)+1 \\ d(i-1,j-1) + \mathbb{1}_{\{l_i \neq k_j\}} \end{array} \right\} \quad (1)$$

Le code est disponible à l'adresse : http://www.xavierdupre.fr/blog/2013-12-02_nojs.html

1) Récupérez la fonction sur Internet et appliquez cette fonction sur les couples de mots suivant :

1. $d(levenstein, levenshtein)$
2. $d(bonbon, bombon)$
3. $d(example, exemples)$
4. $d(esche, eche)$

(1 point)

2) Vérifiez que la distance est symétrique pour ces exemples. (1 point)

3) La fonction donne le même poids à toutes les confusions entre deux caractères. On souhaite que confondre n et m ait un coût de 0,5 au lieu de 1. Que vaut cette nouvelle distance pour la paire $d(bonbon, bombon)$? Modifiez le code de la fonction récupérée à la question 1. (3 points)

4) On veut faire en sorte qu'ajouter (ou supprimer) un s ait un coût de 0.5 ? Modifiez le code de la fonction. Que vaut la nouvelle distance $d(example, exemples)$? (3 points)

5) On veut faire en sorte qu'ajouter (ou supprimer) un s à la fin d'un mot ait un coût de 0,2 ? Que vaut la nouvelle distance $d(example, exemples)$. Modifiez le code de la fonction (de la question précédente). (2 points)

Correction

```
#coding: latin-1

######### énoncé 1, exercice 2 distance de Levenstein

## question 1

def distance_edition(mot1, mot2):
    """
    première fonction retrouvée à : http://www.xavierdupre.fr/blog/2013-12-02_nojs.html
    """
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
```

```
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (1 if c != d else 0)
                opt.append(x)
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

print ("****1*")
print (distance_edition("levenstein","levenshtein"))    # 1
print (distance_edition("bonbon","bonbom"))             # 1
print (distance_edition("example","exemples"))          # 2
print (distance_edition("esche","eche"))                # 1

## question 2

print ("****2*")
print (distance_edition("levenshtein","levenstein"))    # 1
print (distance_edition("bonbom","bonbon"))             # 1
print (distance_edition("exemples","example"))          # 2
print (distance_edition("eche","esche"))                # 1

## question 3

def distance_edition(mot1, mot2):
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                if c == d : x = dist[i-1,j-1]                                          ##
                elif c in ['n','m'] and d in ['n','m'] : x = dist[i-1,j-1] + 0.5    ##
                else : x = dist[i-1,j-1] + 1                                           ##
                opt.append(x)
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

print ("****3*")
print (distance_edition("levenstein","levenshtein"))    # 1
print (distance_edition("bonbon","bonbom"))             # 0.5
print (distance_edition("example","exemples"))          # 2
print (distance_edition("esche","eche"))                # 1
print (distance_edition("levenshtein","levenstein"))    # 1
print (distance_edition("bonbom","bonbon"))             # 0.5
print (distance_edition("exemples","example"))          # 2
print (distance_edition("eche","esche"))                # 1

## question 4

def distance_edition(mot1, mot2):
```

```
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                if c == "s" :  x = dist[i-1,j] + 0.5    ##
                else : x = dist[i-1,j] + 1              ##
                opt.append(x)
            if (i,j-1) in dist :
                if d == "s" : x = dist[i,j-1] + 0.5     ##
                else : x = dist[i,j-1] + 1              ##
                opt.append(x)
            if (i-1,j-1) in dist :
                if c == d : x = dist[i-1,j-1]
                elif c in ['n','m'] and d in ['n','m'] : x = dist[i-1,j-1] + 0.5
                else : x = dist[i-1,j-1] + 1
                opt.append(x)
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

print ("****4*")
print (distance_edition("levenstein","levenshtein"))    # 1
print (distance_edition("bonbon","bonbom"))             # 0.5
print (distance_edition("example","exemples"))          # 1.5
print (distance_edition("esche","eche"))                # 0.5
print (distance_edition("levenshtein","levenstein"))    # 1
print (distance_edition("bonbom","bonbon"))             # 0.5
print (distance_edition("exemples","example"))          # 1.5
print (distance_edition("eche","esche"))                # 0.5

## question 5

def distance_edition(mot1, mot2):
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                if c == "s" :
                    if i == len(mot1)-1 : x = dist[i-1,j] + 0.2 ##
                    else : x = dist[i-1,j] + 0.5                ##
                else : x = dist[i-1,j] + 1
                opt.append(x)
            if (i,j-1) in dist :
                if d == "s" :
                    if j == len(mot2)-1 : x = dist[i,j-1] + 0.2 ##
                    else : x = dist[i,j-1] + 0.5                ##
                else : x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                if c == d : x = dist[i-1,j-1]
                elif c in ['n','m'] and d in ['n','m'] : x = dist[i-1,j-1] + 0.5
                else : x = dist[i-1,j-1] + 1
                opt.append(x)
            dist[i,j] = min(opt)
```

```
    return dist[len(mot1)-1,len(mot2)-1]

print ("****5*")
print (distance_edition("levenstein","levenshtein"))    # 1
print (distance_edition("bonbon","bonbom"))             # 0.5
print (distance_edition("example","exemples"))          # 1.2
print (distance_edition("esche","eche"))                # 0.5
print (distance_edition("levenshtein","levenstein"))    # 1
print (distance_edition("bonbom","bonbon"))             # 0.5
print (distance_edition("exemples","example"))          # 1.2
print (distance_edition("eche","esche"))                # 0.5
```

Quelques remarques :

1. Question 3 : une réponse simple consistait à changer le coût de toutes les comparaisons. Dans ce cas, on ne change pas seulement le coût n/m mais tous les coûts. Ce n'était pas l'effet souhaité ici. J'ai enlevé un point.

   ```
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (0.5 if c != d else 0)  # 1 --> 0.5
   ```

2. Question 4 : la tentation de faire aussi simple ici était la même en changeant le coût de toutes les insertions. J'ai enlevé un point.
3. Question 5 : la tentation était de traiter le cas spécifique ou le `mot1==mot2+"s"` et non pas tous les cas où il y a un "s" final et deux débuts de mot parfois différent. J'ai enlevé un point.

ENSAE TD noté, mardi 27 décembre 2013

Le programme construit au fur et à mesure des questions devra être imprimé à la fin du TD et rendu au chargé de TD. ***Il ne faut pas oublier de mentionner son nom inséré en commentaire au début du programme et de l'ajouter sur chaque page.*** *Les réponses autres que des parties de programme seront insérées sous forme de commentaires. Les squelettes de fonctions proposés ne sont que des suggestions.* ***Il faudra aussi indiquer le numéro des exercices sous forme de commentaires.***

3

On s'intéresse à la distance de Levenshtein qui est une distance entre deux mots. Pour deux mots $L = (l_1, .., l_m)$ et $K = (k_1, ..., k_n)$, elle se résume par la relation de récurrence suivante :

$$d(i,j) = \min \left\{ \begin{array}{l} d(i-1,j)+1 \\ d(i,j-1)+1 \\ d(i-1,j-1) + \mathbb{1}_{\{l_i \neq k_j\}} \end{array} \right\} \tag{2}$$

Le code est disponible à l'adresse : http://www.xavierdupre.fr/blog/2013-12-02_nojs.html

1) Récupérez la fonction sur Internet et appliquez cette fonction sur les couples de mots suivant :

1. $d(levenstein, levenstien)$
2. $d(bonbbon, bonbon)$
3. $d(example, exemples)$

(1 point)

2) Vérifiez que la distance est symétrique pour ces exemples. (1 point)

3) La fonction donne le même poids à toutes les confusions entre deux caractères. On veut modifier la fonction de telle sorte qu'elle donne un coût deux fois moindre aux inversions de lettres. Modifiez le code de la fonction récupérée à la question 1. Quel est le nouveau coût de la paire $d(levenstein, levenstien)$? (4 points)

4) On veut donner à un coût de 0.45 à toutes les répétitions de lettres. Quel est le nouveau coût de la paire $d(bonbbon, bonbon)$? Modifiez le code de la fonction (de la question précédente). (4 points)

Correction

```
#coding: latin-1

######### énoncé 2, exercice 3 distance de Levenstein

## question 1

def distance_edition(mot1, mot2):
    """
    première fonction retrouvée à : http://www.xavierdupre.fr/blog/2013-12-02_nojs.html
    """
    dist = { (-1,-1): 0 }
```

```
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (1 if c != d else 0)
                opt.append(x)
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

print ("****1*")
print (distance_edition("levenstein","levenstien"))     # 2
print (distance_edition("bonbbon","bonbon"))            # 1
print (distance_edition("example","exemples"))          # 2

## question 2

print ("****2*")
print (distance_edition("levenstien","levenstein"))     # 2
print (distance_edition("bonbon","bonbbon"))            # 1
print (distance_edition("exemples","example"))          # 2

## question 3

def distance_edition(mot1, mot2):
    """
    première fonction retrouvée à : http://www.xavierdupre.fr/blog/2013-12-02_nojs.html
    """
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (1 if c != d else 0)
                opt.append(x)
            if (i-2,j-2) in dist :                          ##
                if c == mot2[j-1] and d == mot1[i-1] :      ##
                    x = dist[i-2,j-2] + 1                   ##
                    opt.append(x)                           ##
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

print ("****3*")
print (distance_edition("levenstein","levenstien"))     # 1
```

```
print (distance_edition("bonbbon","bonbon"))            # 1
print (distance_edition("example","exemples"))          # 2
print (distance_edition("levenstien","levenstein"))     # 1
print (distance_edition("bonbon","bonbbon"))            # 1
print (distance_edition("exemples","example"))          # 2

## question 4

def distance_edition(mot1, mot2):
    """
    première fonction retrouvée à : http://www.xavierdupre.fr/blog/2013-12-02_nojs.html
    """
    dist = { (-1,-1): 0 }
    for i,c in enumerate(mot1) :
        dist[i,-1] = dist[i-1,-1] + 1
        dist[-1,i] = dist[-1,i-1] + 1
        for j,d in enumerate(mot2) :
            opt = [ ]
            if (i-1,j) in dist :
                x = dist[i-1,j] + 1
                opt.append(x)
            if (i,j-1) in dist :
                x = dist[i,j-1] + 1
                opt.append(x)
            if (i-1,j-1) in dist :
                x = dist[i-1,j-1] + (1 if c != d else 0)
                opt.append(x)
            if (i-2,j-2) in dist :
                if c == mot2[j-1] and d == mot1[i-1] :
                    x = dist[i-2,j-2] + 1
                    opt.append(x)
            if (i-2,j-1) in dist and c == d == mot1[i-1] :  ##
                    x = dist[i-2,j-1] + 0.45                ##
                    opt.append(x)                           ##
            if (i-1,j-2) in dist and c == d == mot2[j-1] :  ##
                    x = dist[i-1,j-2] + 0.45                ##
                    opt.append(x)                           ##
            dist[i,j] = min(opt)
    return dist[len(mot1)-1,len(mot2)-1]

print ("****4*")
print (distance_edition("levenstein","levenstien"))     # 1
print (distance_edition("bonbbon","bonbon"))            # 0.45
print (distance_edition("example","exemples"))          # 2
print (distance_edition("levenstien","levenstein"))     # 1
print (distance_edition("bonbon","bonbbon"))            # 0.45
print (distance_edition("exemples","example"))          # 2
```

Quelques remarques :

1. Question 3 et 4 : peu ont pensé à ajouter un cas dans la liste `opt`, aucun n'a utilisé les indices $i-2$ et $j-2$. Quelques-uns ont utilisé la différence $i-j$:

   ```
                        if abs(i-j)==1 and mot1[i]==mot2[j] and (mot1[i+1]==mot2[j-1] or mot1[i-1]==mot2[j+1]):
   ```

   Cela marche uniquement pour une inversion à la même position dans les deux mots. J'ai enlevé 0.5 points.

D'autres ont modifié le contenu de la liste `opt` à la fin :

```
                        if abs(j-i)==1 and mot1[i]==mot2[j] and (mot1[i+1]==mot2[j-1] or mot1[i-1]==mot2[j+1]):
                                dist[i,j] = min(opt)-1
                        else:
                                dist[i,j]=min(opt)
```

Cette solution fonctionne parfois mais pas dans tous les cas. Outre quelques problèmes d'indice (`i+1`) lorsqu'on approche la fin d'un mot, cela modifie le coût associée à chaque transformation. Chaque élément dans la liste `opt` correspond à une étape dans la construction du meilleur alignement dans les deux mots. Cela introduit des effets non désirables dans le cas où les deux mots sont `"eeeeeeeee"` et `"eeeefeeeee"`. Certaines réponses ont donné des distances négatives. J'ai enlevé 1 point.

4

La recherche dichotomique est assez simple. On cherche un élément e dans un tableau trié :

1. On compare e à l'élément du milieu.
2. S'ils sont égaux, on s'arrête. Si e est inférieur, on cherche dans la première partie du tableau. On cherche dans la partie supérieure dans l'autre cas.

On part de l'algorithme implémenté ici de façon récursive ou non et disponible ici : http://www.xavierdupre.fr/blog/2013-12-01_nojs.html.

1) Récupérez l'algorithme de la recherche dichotomique et appliquez-le à la liste [0, 2, 3, 5, 10, 100, 340] et le nombre 100. (1 point)

2) Adaptez l'algorithme pour retourner -1 lorsque l'élément à chercher n'est pas dans la liste. Il peut être utile de vérifier que vous arrivez bien à retrouver tous les éléments. (3 points)

```
l = [ 0, 2, 4, 6, 8, 100, 1000 ]
for i in l :
    print (i,recherche_dichotomique(i, l))
```

3) On suppose qu'on a maintenant deux listes triées, il faut écrire une fonction qui cherche un élément dans chacune des deux listes et qui retourne deux positions. (2 points)

```
def deux_recherches(element, liste1, liste2) :
    # ....
    return position_dans_liste1, position_dans_liste2
```

4) On suppose que la liste 1 est 10 fois plus petite que la liste 2. On effectue 1010 recherches. Parmi elles, 1000 nombres sont dans la liste 1, 10 sont dans la liste 2. On considère deux options :

1. On cherche d'abord dans la liste 1. Si rien n'a été trouvé, on cherche dans la liste 2.
2. On cherche dans une liste triée qui contient les deux listes.

Quelle est la plus rapide ? (2 points)

5) On suppose que tous les éléments de la liste 1 sont inférieurs à tous les éléments de la liste 2. Comment utiliser cette information pour ne faire qu'une seule recherche. (2 points)

Correction

```
#coding: latin-1

######### énoncé 1, exercice 1, recherche dichotomique

## question 1

def recherche_dichotomique( element, liste_triee ):
    """
    premier code: http://www.xavierdupre.fr/blog/2013-12-01_nojs.html
    """
    a = 0
    b = len(liste_triee)-1
    m = (a+b)//2
```

```
    while a < b :
        if liste_triee[m] == element :
            return m
        elif liste_triee[m] > element :
            b = m-1
        else :
            a = m+1
        m = (a+b)//2
    return a

l =  [ 0, 2, 3, 5, 10, 100, 340 ]
print (recherche_dichotomique(100, l))  # affiche 5

## question 2

def recherche_dichotomique( element, liste_triee ):
    a = 0
    b = len(liste_triee)-1
    m = (a+b)//2
    while a < b :
        if liste_triee[m] == element :
            return m
        elif liste_triee[m] > element :
            b = m-1
        else :
            a = m+1
        m = (a+b)//2
    if liste_triee[a] != element :  return -1       # ligne ajoutée
    else : return m


l = [ 0, 2, 4, 6, 8, 100, 1000 ]
for i in l :
    print (i,recherche_dichotomique(i, l))
    # vérifier qu'on retrouve tous les éléments existant
print (recherche_dichotomique(1, l))   # affiche -1


## question 3

def deux_recherches(element,liste1,liste2) :
    return recherche_dichotomique(element, liste1) ,  \
            recherche_dichotomique(element,liste2)

l1 = [ 0, 2, 4, 6, 8, 100, 1000 ]
l2 = [ 1200, 3000, 4000, 5555 ]
print (deux_recherches(100,l1,l2) )    # affiche (5,-1)

## question 4

"""
Les logarithmes sont en base 2.

cas 1 : 1000 ln(n) + 10 (ln(n) + ln(10n)) = 1020 ln(n) + 10 ln(10) = C1
cas 2 : 1010 ln(n + 10n) = 1010 ln(n) + 1010 ln(11)               = C2
delta = C2 - C1 = -10 ln(n) + 1010 ln(11) - 10 ln(10)

Conclusion : pour n petit, le cas C1 est préférable, pour les n grands, c'est le cas 2.
"""
```

```
## question 5

def deux_recherches(element,liste1,liste2) :
    if element > liste1[-1] :
        return -1, recherche_dichotomique(element, liste2)
    else :
        return recherche_dichotomique(element,liste1), -1

l1 = [ 0, 2, 4, 6, 8, 100, 1000 ]
l2 = [ 1200, 3000, 4000, 5555 ]
print (deux_recherches(100,l1,l2) )    # affiche (5,-1)
```

Quelques remarques :

1. Question 2 : voir la première remarque associée à la question 4 de l'exercice 1 (page 3).