TD Analyse d'algorithmes : Corrigé

Exercice 1. Réécrire la fonction de recherche simple du cours (dans une liste non triée) pour qu'elle utilise une boucle `while` et non une boucle `for` avec une instruction de sortie (`return`) dans le corps de boucle.

Corrigé. Par exemple.

```
let recherche(L,x):
    i=0
    b=False
    while not b and i<len(L):
        if L[i]==x:
            b=True
        else:
            i+=1
    return b
```

Exercice 2. `truc()` et `bidule()` sont deux fonctions quelconques, sans argument. Déterminer le nombre de fois qu'elles sont appelées dans les scripts ci-dessous.

```
for i in range(n):
    truc()
for i in range(n):
    bidule()
```

```
for i in range(n):
    truc()
    for j in range(n):
        bidule()
```

```
for i in range(n):
    truc()
    for j in range(i):
        bidule()
```

```
for i in range(n):
    truc()
    for j in range(i):
        bidule()
        for k in range(j):
            truc() ; bidule()
```

En supposant que les fonctions s'exécutent en $O(1)$, donner la complexité asymptotique de ces scripts.

Corrigé.

1. truc et bidule s'exécutent n fois chacune. Complexité $O(n)$.
2. truc s'exécute n fois, bidule n^2 fois. Complexité $O(n^2)$.
3. truc s'exécute n fois, bidule $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$ fois. Complexité $O(n^2)$.
4. truc s'exécute $n + \sum_{i=0}^{n-1}\sum_{j=0}^{i-1} j = n + \sum_{i=1}^{n-1} \frac{i(i-1)}{2} = n + \frac{1}{2}\left(\frac{n(n-1)(2n-1)}{6} - \frac{n(n-1)}{2}\right) = \frac{n}{12}\left(2n^2 - 6n + 16\right)$ fois, et bidule $\frac{1}{2}\left(\frac{n(n-1)(2n-1)}{6} + \frac{n(n-1)}{2}\right) = \frac{n(n-1)(2n+2)}{12}$ fois. Complexité $O(n^3)$.

Exercice 3. On considère la fonction suivante :

une fonction f

```
def f(n):
    assert n>=0
    p=1
    for i in range(1,n+1):
        p*=i
    return(p)
```

1. Que fait la fonction f ?

2. Prouver que le résultat est correct en exhibant un invariant de boucle.

3. Quelle est sa complexité en nombre d'opérations arithmétiques ?

Corrigé. Elle calcule la factorielle. Un invariant de la boucle, vrai en haut du corps de boucle, est « p contient $(i-1)!$ ». Sa complexité est clairement de n opérations arithmétiques. Remarque : pour `f(0)`, on ne rentre pas dans la boucle (`range(1,1)` est vide), et la fonction renvoie 1, ce qui est correct.

Exercice 4. Cet exercice fait suite au précédent. On considère la fonction g suivante :

```
def g(n):
    assert n>=0
    s=0
    for i in range(n+1):
        s+=f(i)
    return s
```

1. Que calcule-t-elle ? Exhibez un invariant de boucle.
2. Quelle est sa complexité en nombre d'opérations arithmétiques ?
3. Proposez une idée d'amélioration.

Corrigé. La fonction calcule $\sum_{i=0}^{n} i!$. Un invariant (vrai en haut du corps de boucle) est : « s contient $\sum_{j=0}^{i-1} j!$ » (avec la convention que cette somme est nulle pour $i = 0$). Sa complexité est en $O(n^2)$, car l'appel `f(i)` est en $O(i)$. Une idée d'amélioration est de calculer simultanément factorielle et somme, par exemple :

```
def g2(n):
    s=1 #déja 0!
    p=1
    for i in range(1,n+1):
        p*=i
        s+=p
    return s
```

La complexité est alors $O(n)$.

Exercice 5. *C'est plus, c'est moins, mais dans l'autre sens.* Le but de cet exercice est d'écrire un programme qui joue au « c'est plus c'est moins » en vous posant des questions. Vous choisissez un nombre dans votre tête, supposé être entre 0 et 99. Le programme vous pose des questions du type « est-ce n ? » et vous répondez par « plus » « moins » ou « oui » s'il a gagné.

1. Décrire un algorithme bête qui fonctionnera (on ne demande pas de le coder). Si vous mentez, le programme devra s'en rendre compte. Combien pose-t-il de questions dans le pire cas ? (Le pire cas étant quand vous donnez un maximum de réponses crédibles, mais pas « oui »).
2. Proposez un algorithme inspiré de la recherche dichotomique. Combien de fois pouvez-vous vous permettre de ne pas répondre « oui » sans qu'un mensonge soit détecté ?
3. Codez effectivement l'algorithme.
4. On joue maintenant avec des nombres entre 0 et 100000. Environ combien de questions vous posera-t-il en plus ?

Corrigé.

1. On peut tester tous les nombres entre 0 et 99, dans l'ordre. 100 questions sont posées dans le pire cas !
2. On s'inspire de la recherche dichotomique en maintenant l'invariant : le nombre cherché est entre g inclus et d exclus (initialement, $g = 0$ et $d = 100$). Si $g \geq d$, il y a eu mensonge ! À chaque fois, le nombre d'entiers possibles est divisé par 2, et $100 < 2^7 = 128$. Ainsi on peut répondre six fois « plus » ou « moins » au maximum, au septième le mensonge est détecté.
3. Par exemple :

```
def plus_moins():
    g=0
    d=100
    while g<d:
        m=(g+d)//2
        a=input("est-ce "+str(m)+" ? ")
        if a=="oui":
            print("j'ai gagné !")
            return
        elif a=="plus":
            g=m+1
        else:
            d=m
    print("tu triches !")
```

Exemple de déroulement :

```
>>> plus_moins()
est-ce 50 ? plus
est-ce 75 ? moins
est-ce 63 ? plus
est-ce 69 ? moins
est-ce 66 ? plus
est-ce 68 ? moins
est-ce 67 ? plus
tu triches !
```

Rappel : `return` seul permet d'interrompre une fonction sans renvoyer de résultat. `input` affiche un message d'invite facultatif, et récupère ce que l'utilisateur entre au clavier sous forme de chaîne de caractères.

4. $\log_2(10000) < 14$. Donc le programme posera au plus 14 questions, soit 7 de plus seulement.

Exercice 6. *Plus petit entier naturel non présent dans une liste.* On s'intéresse ici au problème de déterminer le plus petit entier naturel non présent dans une liste constituée d'entiers.

1. Écrire une fonction `est_present(L,x)` renvoyant un booléen indiquant si `x` est présent dans la liste `L`.

2. En déduire une fonction `ppenp(L)` renvoyant le plus petit entier naturel non présent dans `L`. On utilisera une boucle `while`.
3. Estimer sa complexité dans le pire cas en fonction de la taille n de la liste passée en entrée.
4. Proposez une solution de complexité linéaire en la taille de la liste. *Indication : utiliser une liste de booléens de taille n ou $n+1$.*

Corrigé.

1. C'est du cours !
2.
```
def ppenp(L):
    i=0
    while est_present(L,i):
        i+=1
    return i
```

3. Notons n la taille de la liste. Dans le pire cas tous les entiers de 0 à $n-1$ sont présents, donc le plus petit non présent est n. `est_present` est de complexité $O(n)$, donc on a une complexité totale $O(n^2)$.
4. Voici une solution avec une liste de booléens de taille $n+1$: initialement, tous les `B[i]` sont faux. Ensuite on parcourt la liste, si un élément est entre 0 et n on « coche » dans la liste le `B[i]` correspondant. Comme au plus $n-1$ éléments sont dans la liste initiale, l'un des éléments entre 0 et n n'est pas coché : on renvoie le plus petit. La complexité est donc $O(n)$.

```
def ppenp2(L):
    n=len(L)
    B=[False]*(n+1)
    for x in L:
        if 0<=x<=n:
            B[x]=True
    for i in range(n+1):
        if not B[i]:
            return i
```

Exercice 7. *Tri par comptage.* On s'intéresse au problème du tri d'une liste d'entiers naturels, constituée d'entiers strictement inférieurs à une borne k connue. Le tri par sélection déja vu en cours est de complexité $O(n^2)$ avec n la taille de la liste. Si k est petit, on peut donner une solution plus efficace.

1. On suppose dans cette question que $k=2$. Écrire une fonction `tri_bits(L)` prenant en entrée une liste constituée de 0 et de 1 et renvoyant une nouvelle liste, équivalente à `L`, triée. Comme le nom de l'exercice l'indique, on procédera par comptage.
2. Quelle est la complexité en fonction de la taille n de la liste ?
3. Donner une fonction `tri_comptage(L,k)` basée sur le même principe, prenant également en entrée un entier k, la liste étant supposée contenir uniquement des entiers entre 0 et $k-1$. *Indication : utiliser une liste d'entiers de taille k.*
4. Estimer sa complexité en fonction de n et k.

Corrigé.

1.
```
def tri_bits(L):
    nb0, nb1 = 0, 0 #nombre de zéros, de uns
    for x in L:
        if x==0:
            nb0+=1
        else:
            nb1+=1
    return [0]*nb0 + [1]*nb1 #concaténation de nb0 zéros, avec nb1 uns.
```

2. La complexité est linéaire en la taille de la liste.
3. On suit la même idée, sauf que pour compter on utilise une liste de taille k (on ne peut pas déclarer k variables !)

```
def tri_comptage(L, k):
    NB=[0]*k #NB[0], nombre de zéros, NB[1] nombre de uns, etc..
    for x in L:
```

```
        NB[x]+=1
    L2=[]
    for i in range(k):
        for j in range(NB[i]):
            L2.append(i) #on rajoute NB[i] fois l'élément i.
    return L2
```

Remarque : plutôt que la boucle interne, on peut utiliser `L2.extend([i]*NB[i])` ou `L2+=[i]*NB[i]`. La complexité est la même.

4. La complexité est $O(n+k)$. Explications : $O(k)$ pour créer `NB`, $O(n)$ pour parcourir `L`, et $O(n+k)$ pour construire `L2`. *Remarque :* voici un exemple où la majoration nombre de tours $\times$ complexité d'un tour donne une réponse surestimée ($O(nk)$) à la complexité. Ici, il faut bien comprendre que l'instruction `L2.append(i)` est exécutée n fois au total.

Exercice 8. *Recherche de pic dans une liste*. On dit qu'un élément d'une liste est un pic s'il est supérieur 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 51, 78 et 82. Cet exercice a pour but de trouver les indices des pics dans une liste : les indices des pics précédents sont 1, 4 et 6.

1. Écrire une fonction `indices_pics(L)` prenant en entrée une liste (qu'on pourra éventuellement supposer de taille au moins 2) et renvoyant la liste des indices de ses pics.

Les questions qui suivent ont pour but d'écrire une fonction cherchant l'indice d'*un seul pic*, mais de manière plus efficace que la fonction précédente.

2. Soit $i \geq 1$ un indice de la liste. On suppose que `L[i]<L[i-1]`. Montrer que les pics de la liste `L[:i]` (liste constituée des éléments de `L` d'indice entre 0 et $i-1$) sont exactement ceux de `L` situés avant l'indice i exclus. Justifier que la liste `L[:i]` contient au moins un pic.

3. En vous inspirant de la question précédente et de la recherche dichotomique, écrire une fonction 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).

Corrigé.

1. Dans la fonction suivante, on regarde si `L[i]` est plus grand que le précédent et le suivant, *s'ils existent !*

```
def indices_pics(L):
    n=len(L)
    P=[]
    for i in range(n):
        if (i==0 or L[i]>=L[i-1]) and (i==n-1 or L[i]>=L[i+1]):
            P.append(i)
    return P
```

2. Le seul élément qui pourrait être un pic de `L[:i]` sans être un pic de `L` est `L[i-1]`. Mais il est plus grand que `L[i]`, donc si c'est un pic dans `L[:i]`, c'en est un dans `L` également. `L[:i]` est non vide donc contient au moins un pic : son maximum !

3. Voici :

```
def indice_pic(L):
    g=0
    d=len(L)
    while True:
        """ invariant de boucle: il y a un pic d'indice i avec g<=i<d """
        m=(g+d)//2
        if m>0 and L[m]<L[m-1]: #il y a un pic à gauche
            d=m
        elif m<len(L)-1 and L[m]<L[m+1]: #il y a un pic à droite
            g=m+1
        else: #m est l'indice d'un pic !
            return m
```

Exercice 9. *Calcul de* $\lfloor \sqrt{n} \rfloor$. On considère la fonction suivante, prenant en entrée un entier au moins égal à 1.

```
def racine_int(n):
    i=1
    s=n
    while s-i>1:
        m=(i+s)//2
        if m*m<=n:
            i=m
        else:
            s=m
    return i
```

1. Montrer que $s-i$ est un *variant de boucle* (une quantité qui diminue strictement à chaque passage dans la boucle).
2. En déduire la terminaison de l'algorithme.
3. Montrer que $i^2 \leq n < s^2$ est un invariant de la boucle (pour $n > 1$).
4. En déduire la correction de l'algorithme.
5. Quelle est sa complexité arithmétique (nombre d'opérations effectuées) ?

Corrigé.

1. Effectuons un passage dans la boucle : cela signifie que $s - i \geq 2$. On note s' et i' les valeurs de s et i en bas du corps de boucle. Comme $s - i \geq 2$, on a $i < m < s$. En bas du corps de boucle, on a $(i', s') \in \{(i, m), (m, s)\}$. Dans les deux cas $s' - i' < s - i$. Ainsi, $s - i$ est une *quantité*, à valeurs dans $\mathbb{N}$, qui décroît *strictement* à chaque passage dans la boucle : c'est un variant.
2. Un variant suffit à montrer qu'une boucle `while` termine !
3. Supposons $n > 1$. On a alors $i^2 \leq n < s^2$ vrai avant la boucle. Supposons la propriété vraie en haut du corps de boucle, et notons encore i' et s' les valeurs de i et s en bas du corps de boucle :
   - si $m^2 \leq n$, on a $(i', s') = (m, s)$, et donc $i'^2 \leq n < s'^2 = s^2$ et la propriété est vraie en bas du corps de boucle.
   - sinon, $m^2 > n$, on a $(i', s') = (i, m)$, et donc $i^2 = i'^2 \leq n < s'^2$ et la propriété est aussi vraie en bas du corps de boucle.

   C'est un invariant de boucle !
4. On veut montrer que l'algorithme renvoie bien $\lfloor \sqrt{n} \rfloor$, pour $n \geq 1$.
   - si $n = 1$, on ne rentre pas dans la boucle, et l'algorithme renvoie 1 : c'est correct ;
   - sinon, on sait qu'après la boucle $i^2 \leq n < s^2$ (l'invariant est conservé) et $s - i \leq 1$ (la condition de continuation de la boucle est fausse). Ainsi $s \neq i$ car $i^2 < s^2$, donc $s = i + 1$. L'invariant donne alors $i^2 \leq n < (i+1)^2$, d'où $i \leq \sqrt{n} < i + 1$, donc $i = \lfloor \sqrt{n} \rfloor$. L'algorithme renvoie donc bien $\lfloor \sqrt{n} \rfloor$.
5. Le schéma est le même que l'algorithme dichotomique : la quantité $s - i$ est diminuée de moitié à chaque étape. On a donc une complexité $O(\log n)$.

Exercice 10. *Triplets Pythagoriciens.* Soit $N > 0$. On veut calculer tous les triplets d'entiers (a, b, c) tels que $1 \leq a \leq b < c \leq N$ et $a^2 + b^2 = c^2$. On ne travaillera qu'avec des entiers dans cet exercice (pas de flottants)

1. Proposer un algorithme naïf utilisant trois boucles imbriquées, renvoyant la liste des triplets convenables.
2. Quelle est sa complexité ?
3. En utilisant la fonction de l'exercice précédent, proposer une méthode de meilleure complexité.

Corrigé.

1. Par exemple :

```
def triplets(N):
    L=[]
    for a in range(1,N):
        for b in range(a,N):
            for c in range(b+1,N+1):
                if a*a+b*b==c*c:
                    L.append((a,b,c))
    return L
```

2. La complexité est $O(N^3)$.

3. La fonction suivante convient également : il suffit de tester si a^2+b^2 est un entier.

```
def triplets(N):
    L=[]
    for a in range(1,N):
        for b in range(a,N):
            c=racine_int(a*a+b*b)
            if c*c==a*a+b*b:
                L.append((a,b,c))
    return L
```

L'appel à `racine_int` se fait sur un entier inférieur à $2N^2$, et a donc une complexité $O(\log(2N^2)) = O(\log N)$. On a donc une complexité totale $O(N^2 \log N)$.