Correction du DS d'Informatique n°4

Ce sujet est composé de deux exercices et un problème. Vous prendrez soin de bien justifier vos calculs.
Veillez à bien respecter l'indentation des programmes Python que vous écrirez sur votre copie.

Exercice 1. *CCP 2016*

Les algorithmes demandés doivent être écrits en Python. On sera très attentif à la rédaction et notamment à l'indentation du code. Cet exercice étudie deux algorithmes permettant le calcul du pgcd (plus grand commun diviseur) de deux entiers naturels.

I.1. Pour calculer le pgcd de 3705 et 513, on peut passer en revue tous les entiers $1,2,3,\cdots,512,513$ puis renvoyer parmi ces entiers le dernier qui divise à la fois 3705 et 513. Il sera alors bien le plus grand des diviseurs communs à 3705 et 513. Écrire une fonction `gcd` qui renvoie le pgcd de deux entiers naturels non nuls, selon la méthode décrite ci-dessus. On pourra éventuellement utiliser librement l'instruction `min(a,b)` qui calcule le minimum de a et b. Par exemple `gcd(3705, 513)` renverra 57.

I.2. L'algorithme d'Euclide permet aussi de calculer le pgcd. Voici une fonction Python nommée `euclide` qui implémente l'algorithme d'Euclide.

```
def euclide(a,b):
    """Donn\'ees: a et b deux entiers naturels
       R\'esultat: le pgcd de a et b, calcul\'e par l'algorithme d'Euclide"""
    u = a
    v = b
    while v != 0:
        r = u % v
        u = v
        v = r
    return u
```

Écrire une fonction «récursive» `euclide_rec` qui calcule le pgcd de deux entiers naturels selon l'algorithme d'Euclide.

I.3. On note $(F_n)_{n\in\mathbb{N}}$ la suite des nombres de Fibonacci définie par :

$$F_0 = 0, F_1 = 1, \quad \forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n.$$

I.3.a. Écrire les divisions euclidiennes successivement effectuées lorsque l'on calcule le pgcd de $F_6 = 8$ et $F_5 = 5$ avec la fonction `euclide`.

I.3.b. Soit $n \geq 2$ un entier. Quel est le reste de la division euclidienne de F_{n+2} par F_{n+1} ? On pourra utiliser librement que la suite $(F_n)_{n\in\mathbb{N}}$ est strictement croissante à partir de $n = 2$. En déduire, sans démonstration, le nombre u_n de divisions euclidiennes effectuées lorsque l'on calcule le pgcd de F_{n+2} et F_{n+1} avec la fonction `euclide` .

I.3.c. Comparer pour n au voisinage de $+\infty$, ce nombre u_n, avec le nombre v_n de divisions euclidiennes effectuées pour le calcul du pgcd de F_{n+2} et F_{n+1} par la fonction `gcd`. On pourra utiliser librement que F_n est équivalent, au voisinage de $+\infty$, à $\phi^n/\sqrt{5}$ où $\phi = (1+\sqrt{5})/2$ est le nombre d'or.

I.4. Écrire une fonction `fibo` qui prend en argument un entier naturel n et renvoie le nombre de Fibonacci F_n. Par exemple, `fibo(6)` renverra 8.

I.5. En utilisant la fonction `euclide`, écrire une fonction `gcd_trois` qui renvoie le pgcd de trois entiers naturels. Par exemple, `gcd_trois(18, 30, 12)` renverra 6.

Correction.

I.1.
```
def gcd(a,b):
    """Donn\'ees: aet b deux entiers naturels
       R\'esultat: le pgcd de a et b, calcul\'e par l'algorithme brut de force"""
    n=min(a,b)
    res=1
    for i in range(1,n+1):
        if a%i==0 and b%i==0: #si i divise a et b
            res=i
    return res
```

I.2. Une fonction « récursive » `euclide_rec` qui calcule le pgcd de deux entiers naturels :
```
def euclide_rec(a,b):
    """Donn\'ees: a et b deux entiers naturels valable même si a ou b est nul
       R\'esultat: le pgcd de a et b, calcul\'e par l'algorithme d'Euclide récursifi\'e"""
    if b==0:
        return a
    r=a%b
    return euclide_rec(b,r)
```

I.3.a. On a les divisions euclidiennes successives où les quotients valent tous 1 :

i : $F_6 = 1 \times F_5 + F_4$ et $F_4 < F_5$ donc de reste $F_4 = 3$
ii : $F_5 = 1 \times F_4 + F_3$ et $F_3 < F_4$ donc de reste $F_3 = 2$
iii : $F_4 = 1 \times F_3 + F_2$ et $F_2 < F_3$ donc de reste $F_2 = 1$
iv : $F_3 = 1 \times F_2 + F_1$ et $F_1 < F_2$ donc de reste $F_1 = 1$
v : $F_2 = 1 \times F_1 + F_0$ et $F_0 < F_1$ donc de reste $F_0 = 0$

I.3.b. On a $F_{n+2} = 1 \times F_{n+1} + F_n$ et $F_n < F_{n+1}$ car la suite $(F_n)_{n\in\mathbb{N}}$ est strictement croissante à partir de $n = 2$

$\boxed{\text{le reste de la division euclidienne de } F_{n+2} \text{ par } F_{n+1} \text{ est } F_n}$

$\boxed{\text{le nombre de divisions euclidiennes effectuées pour le calcul de } \texttt{euclide}(F_{n+2}, F_{n+1}) \text{ est } u_n = n+1}$

I.3.c. Dans l'appel de `gcd`(F_{n+2}, F_{n+1}), à chaque tour de boucle, sont effectués deux divisions euclidiennes. Le nombre de tour de boucles est $\min(F_{n+2}, F_{n+1}) = F_{n+1}$ donc

$\boxed{\text{le nombre de divisions euclidiennes effectuées pour le calcul de } \texttt{gcd}(F_{n+2}, F_{n+1}) \text{ est } v_n = 2F_{n+1}}$

donc $\boxed{v_n \underset{n\to+\infty}{\sim} \frac{2\phi^{u_n}}{\sqrt{5}}}$ comme on s'en doutait la méthode brute est moins bonne en terme de complexité

I.4. Plusieurs fonctions `fibo` qui prend en argument un entier naturel n et renvoie le nombre de Fibonacci F_n :
```
def fibo(n):
    """Donn\'ees: n entier naturel
       R\'esultat: le terme de rang n de la suite de Fibonnacci
       sans contrainte de complexité !!! (faut aller vite en respectant la consigne !)"""
    if n<2:
        return n
    return fibo(n-1) + fibo(n-2)
def fibo2(n):
    """Donn\'ees: n entier naturel
       R\'esultat: le terme de rang n de la suite de Fibonnacci
       version linéaire non récursive (préférable ?)"""
    if n<2:
```

```
        return n
    u=0
    v=1
    for i in range(n-1):
        w=v
        v=u+v
        u=w # sans pythonnerie u,v=v,u+v
    return v
```

I.5. Une fonction `gcd_trois` qui renvoie le pgcd de trois entiers naturels $a \wedge b \wedge c = c \wedge (a \wedge b)$:

```
def gcd_trois(a,b,c):
    return euclide(c,euclide(a,b))
```

Exercice 2. *Algorithmes dichotomiques*

1. Programmattion de l'algorithme de recherche dichotomique.
   Dans la suite, E est un ensemble totalement ordonné, `L` est un tableau **trié** d'éléments de E et `x` est un élément de E.
   (a) Écrire une fonction `recherche(L,x)` qui opère une recherche de complexité linéaire de `x` dans `L`.
   (b)

   > **Principe de la recherche dichotomique :**
   > On considère un tableau `L` que l'on suppose **triée** on suppose donc que ses éléments appartiennent à un ensemble muni d'un (pré-)ordre total et on considère un élément `x` que l'on se propose de chercher dans `L`.
   >
   > — Si `L` est vide, `x` n'est pas dans `L`.
   > — Si `L` n'est pas vide, on "coupe" le tableau `L` en deux en son milieu (ou presque) et on considère l'élément `l` à cet endroit :
   > — si `l` vaut `x`, alors `x` est dans `L` ;
   > — si `l` est strictement plus petit que `x`, alors, `L` étant triée, `x` est potentiellement dans la moitié droite de `L` ;
   > — si `l` est strictement plus grand que `x`, alors, `L` étant triée, `x` est potentiellement dans la moitié gauche de `L` ;
   > — Dans les deux derniers cas, on répète l'algorithme sur la moitié qui contient potentiellement `x`.

   En vous basant sur la description ci-dessus, écrire une fonction `dicho_rec(L,x)` récursive de la recherche dichotomique de `x` dans `L`.
   (c) Écrire une fonction `dicho_it(L,x)` itérative de la recherche dichotomique de `x` dans `L`.
2. Programmation de l'algorithme d'exponentiation rapide.
   Dans la suite, `x` est un nombre et `n` est un entier positif. On n'aura bien-sûr jamais recours à l'opérateur `**` dans les questions qui suivent !
   (a) Écrire une fonction `puissance(x,n)` qui renvoie la valeur de `x` à la puissance `n` de complexité linéaire.

(b)

> **Principe de l'exponentiation rapide :**
> On considère un nombre x et un entier naturel n.
>
> — Si n vaut 0, on renvoie 1.
> — Si $n \geq 1$:
> — si n est pair, on a $x^n = \left(x^{\frac{n}{2}}\right)^2$;
> — si n est impair, on a $x^n = x\left(x^{\frac{n-1}{2}}\right)^2$;
> — puis on répète l'algorithme pour le calcul de $x^{\frac{n}{2}}$ ou de $\left(x^{\frac{n-1}{2}}\right)$.

En vous basant sur la description ci-dessus, écrire une fonction `expo_rec(x,n)` récursive de l'exponentiation rapide.

(c) Écrire une fonction `expo_it(x,n)` itérative de l'exponentiation rapide.

Correction.

1. (a)

```python
def recherche(L,x):
    for e in L:
        if x == e:
            return True
    return False
```

(b)

```python
### 1ere version avec slicing, peu économique en mémoire
def dicho_rec(L,x):
    if len(L)==0:
        return False
    else:
        m = len(L)//2
        if x == L[m]:
            return True
        elif x < L[m]:
            return dicho_rec(L[:m],x)
        else:
            return dicho_rec(L[m+1:],x)

#2eme version, plus économe en mémoire
def dicho_rec_ind(L,x,deb,fin):
    if deb>=fin:
        return False
    else:
        m = (deb+fin)//2
        if x == L[m]:
            return True
        elif x < L[m]:
            return dicho_rec_ind(L,x,deb,m)
        else:
            return dicho_rec_ind(L,x,m+1,fin)

def dicho_rec2(L,x):
    return dicho_rec_ind(L,x,0,len(L))
```

(c)

```python
def dicho_it(L,x):
    deb = 0
    fin = len(L)
    while deb<fin:
        m = (deb+fin)//2
        if x == L[m]:
            return True
        elif x < L[m]:
            fin = m
        else:
            deb = m+1
    return False
```

2. (a)

```python
def puissance(x,n):
    p=1
    for i in range(1,n+1):
        p = p*x #ou p*=x
    return p
```

(b)

```python
def expo_rec(x,n):
    if n == 0:
        return 1
    else:
        p = expo_rec(x,n//2)
        if n%2 == 0:
            return p*p
        else:
            return p*p*x
```

(c)

```python
def expo_it(x,n):
    N = n
    X = x
    p=1
    while N > 0:
        if N%2 == 1:
            p = p*X
        X = X*X
        N = N//2
    return p
```