TD D'INFORMATIQUE 1
Analyse de programme : terminaison et correction
Corrigé

1 Recherche par clé

1. Écrire une fonction `indice` prenant en argument une liste `L` et un élément `x`, et renvoyant un indice `i` tel que `L[i]` = `x`, ou `None` si `x` n'apparaît pas dans `L`.

   Corrigé :

```
def indice(L,x):
    n = len(L)
    for i in range(n):
        if x == L[i]:
            return i
    return None
```

2. Étudier la terminaison et la correction de la fonction précédente.

   Corrigé :

   La terminaison est évidente puisqu'il n'y a qu'une boucle `for`.

   Pour la correction, on pose I : `x` n'apparait pas dans `L[0:i]`.

   - Au début de la boucle, `i` vaut 0, et `L[0:i]` vaut donc `[]` et I est vérifié.
   - Considérons une itération pour laquelle I est vrai au début. On note avec ' les valeurs en fin d'itération. On a `i' = i+1`. Si l'itération est interrompue par le `return`, on observe que la spécification est bien vérifiée, puisque `L[i]` = `x`. Sinon, on en déduit que `x` n'apparait ni dans `L[0:i]` (d'après I), ni dans L[i], donc n'apparait pas dans `L[0:i+1]` = `L[0:i']`.
   - En conclusion, I reste vrai à la fin de chaque itération, en particulier en sortie de boucle. On a alors $i = n$ (car au début de la dernière itération, $i = n - 1$), et donc `x` n'apparait pas dans `L[0:n]`, c'est-à-dire dans `L`. Le `return None` est donc bien correct.

2 Analyse d'une fonction récursive

On considère la fonction suivante :

```
def FF(a,b):
    assert b > a
    if b == a+1:
        return b
    else :
        c = (a+b)//2
        return FF(a,c) * FF(c,b)
```

1. Calculer `FF(3,8)`. En déduire une spécification pour cette fonction. En particulier, quelle conjecture peut-on proposer pour la valeur renvoyée par `FF(0,n)` ?

   Corrigé :

   `FF(3,8)` = `FF(3,5).FF(5,8)`
   = (`FF(3,4).FF(4,5)`.(`FF(5,6).FF(6,8)`
   = (4.5).(6.(`FF(6,7).FF(7,8)`))
   = (4.5).(6.(7.8))

   On conjecture donc : `FF(a,b)` $= \prod_{a+1}^{b} k$, en particulier `FF(0,n)` $= n!$

2. Démontrer la terminaison et la correction (relativement à la spécification proposée précédemment) de cette fonction.
   Corrigé :

   On pose, pour $n \in \mathbb{N}^*$, P_n : $\forall a, b \in \mathbb{N}$, si $b - a = n$, alors `FF(a,b)` termine et renvoie $\prod_{a+1}^{b} k$.

   On montre $\forall n \in \mathbb{N}^*, P_n$ par récurrence forte sur n.
   - Initialisation : Soient $a, b \in \mathbb{N}$ tels que $b - a = 1$. Par lecture de la fonction, `FF(a,b)` termine et renvoie $b = \prod_{b}^{b} k$.
   - Hérédité : Soit $n > 1 \in \mathbb{N}$ tel que $\forall k \in [\![1, n-1]\!], P_k$. Soient $a, b \in \mathbb{N}$ tels que $b - a = n$. Soit $c = \left\lfloor \frac{a+b}{2} \right\rfloor$. On a donc $\frac{a+b}{2} - 1 < c \leq \frac{a+b}{2}$. On a :

   $$c - a \leq \frac{a+b}{2} - a = \frac{b-a}{2} < b - a = n$$

   $$b - c < b - \frac{a+b}{2} + 1 = \frac{b-a}{2} + 1 = \frac{n-2+2}{2} + 1 \leq n - 2 + 1 + 1 = n$$

   On en déduit par hypothèse de récurrence que les appels récursifs `FF(a,c)` et `FF(c,b)` terminent et sont corrects. L'appel `FF(a,b)` termine donc lui aussi et renvoie

   $$\prod_{a+1}^{c} k \times \prod_{c+1}^{b} k = \prod_{a+1}^{c} k$$

   .

3 Algorithme d'Euclide étendu

Pour $(a, b) \in \mathbb{N}^2$, le théorème de Bézout énonce que l'équation $au + bv = \texttt{pgcd}(a, b)$, où u et v sont des inconnues dans $\mathbb{Z}$, a au moins une solution (et donc une infinité).

L'algorithme d'Euclide étendu permet de calculer une telle solution, en plus du pgcd :

```
def pgcd_bezout(a,b):
    r0, r1 = a, b
    u0, u1 = 1, 0
    v0, v1 = 0, 1
    while r1 != 0:
        q = r0 // r1
        r0, r1 = r1, r0-q*r1
        u0, u1 = u1, u0-q*u1
        v0, v1 = v1, v0-q*v1
    return (r0, u0, v0)
```

1. Exécuter à la main cette fonction sur l'entrée $(14, 49)$. Préciser à quoi correspond le résultat.
   Corrigé : Le programme renvoie le triplet $(7, -3, 1)$. Cela signifie que le pgcd de 14 et 49 vaut 7, et que $7 = -3 \times 14 + 1 \times 49$.
2. Démontrer la terminaison de cette fonction.
   Corrigé : On utilise pour variant `r1`. Par lecture du programme, on observe qu'à chaque étape, `r1` est remplacé par `r0-q*r1`, qui vaut `r0%r1`. On en déduit que `r1` reste positif et décroît strictement, ce qui garantit la terminaison.
3. Démontrer la correction de cette fonction. On pourra admettre :

   $$\forall x \in \mathbb{N},\ \forall y \in \mathbb{N}^*.\quad \texttt{pgcd}(x, y) = \texttt{pgcd}(y, x\%y)$$

Corrigé :
On utilise comme invariant la conjonction de :

$$(\texttt{pgcd}(a,b) = \texttt{pgcd}(r0,r1)) \quad (1)$$

$$(a*u0 + b*v0 = r0) \quad (2)$$

$$(a*u1 + b*v1 = r1) \quad (3)$$

- En entrée de boucle, on vérifie aisément que chaque propriété est vérifiée.
- On considère une itération quelconque, on suppose que l'invariant est vrai au début de cette itération.
  Les valeurs des variables en fin d'itération sont :
  $r0' = r1$
  $r1' = r0 - qr1$
  $u0' = u1$
  $u1' = u0 - qu1$
  $v0' = v1$
  $v1' = v0 - qv1$
  où $q = r0//r1$.

  On a alors
  $\texttt{pgcd}(r0', r1') = \texttt{pgcd}(r1, r0\%r1) = \texttt{pgcd}(r0,r1) = \texttt{pgcd}(a,b)$, donc (1') est vérifié.

  $au0' + bv0' = au1 + bv1 = r1 = r0'$ donc (2') est vérifiée.

  $au1' + bv1' = a(u0 - qu1) + b(v0 - qv1) = au0 + bv0 - q(au1 + bv1) = r0 - qr1 = r1'$, donc (3') est vérifiée.
- En sortie de boucle, $r1 = 0$.
  D'après (1), on a donc $r0 = pgcd(r0, 0) = pgcd(a,b)$.
  D'après(2), $r0 = au0 + bv0$.
  Le triplet $(r0, u0, v0)$ vérifie donc bien les propriétés désirées.

4 Exponentiation rapide

.
On rappelle que l'exponentiation rapide est un algorithme permettant de calculer efficacement x^n en utilisant les décompositions $x^{2k} = x^k * x^k$ et $x^{2k+1} = x * x^k * x^k$.

1. Écrire une fonction récursive implémentant cet algorithme.
   Corrigé :

```
def exporapide(x,n):
    if n==0:
        return 1
    elif n%2==0:
        y = exporapide(x,n//2)
        return y*y
    else:
        y = exporapide(x,n//2)
        return y*y*x
```

2. Démontrer la terminaison et la correction de cette fonction.
   Corrigé :
   On pose, pour $n \in \mathbb{N}$, P_n : `exporapide(x,n)` termine et renvoie x^n. On montre $\forall n \in \mathbb{N}, P_n$ par récurrence forte sur n.

- Initialisation : `exporapide(x,n)` termine et renvoie 1.
- Hérédité : Soit $n \in \mathbb{N}^*$ tel que $\forall k \in [\![1, n-1]\!], P_k$. Soit $q = \left\lfloor \frac{n}{2} \right\rfloor$. On a bien $q \in [\![1, n-1]\!]$, donc l'appel récursif `exporapide(x,n//2)` termine et renvoie x^q. Si n est pair, on a bien $x^n = (x^q)^2 = y^2$, et si n est impair, $x^n = x^{2q+1} = y^2.x$.

5 Tri par sélection

Le tri par sélection consiste à chercher le minimum et à la ranger à sa place, puis chercher le deuxième plus petit élément et le ranger à sa place et ainsi de suite. Lors du TP8, nous avons proposé le code suivant pour ce tri :

```
def minimum(L,deb):
    '''
    Prend en argument une liste et un indice et
    renvoie le plus petit element de la partie de la liste
    entre l'indice deb et la fin de la liste
    '''
    p, m = deb, L[deb]
    for i in range(deb+1, len(L)):
        if L[i]<m:
            p, m = i, L[i]
    return p,m

def tri_selection(L):
    n = len(L)
    for i in range(0,n-1):
        p, m =minimum(L,i)
        L[i], L[p] = L[p], L[i]
```

Étudier la terminaison et la correction de ces deux fonctions.
Corrigé :
On note qu'après k passages dans la boucle de la fonction `tri_selection` , les k premiers éléments de la liste sont à leur place définitive.
Terminaison : La terminaison est immédiate car chacune des fonctions contient une boucle `for`.
Correction : Pour la boucle dans la fonction `minimum` on peut utiliser l'invariant : « m et p sont la valeur minimale et l'indice auquel elle est atteinte, de la sous-liste constituée des éléments entre les indices `deb` et $i-1$ ».
Pour la boucle dans la fonction `tri_selection` on peut utiliser l'invariant : « la liste est une permutation de la liste initiale, et la sous-liste des indices entre 0 et $i-1$ est triée et tous les éléments situés aux indices supérieurs ou égaux à i sont plus grand que `L[i-1]` ».

6 Évaluation de polynômes

Dans cet exercice, on représente un polynôme $a_0 + a_1X + \ldots + a_nX^n$ par la liste `[`$a_0, a_1, \ldots, a_n$`]`

1. Écrire une fonction `evaluer` prenant en argument une liste de flottants `L` et un flottant `x`, et renvoyant la valeur en `x` du polynôme représenté par `L`.
   Corrigé :

   ```
   def evaluer(L,x):
       r = 0
       xp = 1
       for i in range(len(L)):
           r =  r + L[i]*xp
           xp = xp*x
       return r
   ```

2. Que calcule la fonction suivante ?

```
def horner(L,x):
    r=0
    n=len(L)
    for i in range(n-1,-1,-1):
        r=r*x+L[i]
    return r
```

Corrigé : On vérifie sur des tests qu'elle calcule également la valeur du polynôme représenté par L en x.

3. Démontrer la correction de cette fonction.

Corrigé :

On utilise comme invariant :

$$r = \sum_{k=i+1}^{n-1} L[k] * x^{k-(i+1)}$$

- En début de boucle, $r = 0$ et $i = n - 1$. On a bien $0 = 0$ (la somme est vide).
- On considère une itération quelconque et on suppose que l'invariant est vrai au début, ie $r = \sum_{k=i+1}^{n-1} L[k] * x^{k-(i+1)}$.

  Les valeurs des variables en fin d'itération sont :

  $r' = rx + L[i]$

  $i' = i - 1$

  On calcule alors

$$r' = rx + L[i] = \left(\sum_{k=i+1}^{n-1} L[k] * x^{k-(i+1)} \right) x + L[i]$$

$$= \sum_{k=i+1}^{n-1} L[k] * x^{k-i} + L[i] = \sum_{k=i}^{n-1} L[k] * x^{k-i} = \sum_{k=i'+1}^{n-1} L[k] * x^{k-(i'+1)}$$

  L'invariant est donc vrai à la fin de l'itération.
- En sortie de boucle, $i = -1$ et l'invariant est vrai, d'où :

$$r = \sum_{k=0}^{n-1} L[k] * x^k$$

  r contient donc bien la valeur de $P(x)$.