Devoir d'informatique n° 3 (2 heures)

Ce devoir est constitué de plusieurs petits exercices. L'ordre des exercices ne correspond à aucun critère de difficulté ou de longueur : vous pouvez les traiter dans l'ordre que vous voulez. Veillez à soigner la copie tant pour l'écriture, la propreté que pour la rédaction, la rigueur et l'argumentation. De plus, on prêtera une attention particuliére au respect des alignements et des indentations des séquence d'instructions Python. La calculatrice est interdite.
Vous numéroterez vos copies et ferez apparaître clairement sur la première page le nombre de copies.
Consignes particulières : Lorsque l'on demande d'écrire une fonction classique d'opérations sur les listes ou tout autre objet, et qu'une telle fonction ou méthode existe, vous n'êtes évidemment pas autorisé à l'utiliser

Exercice 1 : Décomposition binaire

1. Ecrire une fonction **DecVersBin** prenant comme argument un entier naturel n et retournant la liste des bits constituant l'écriture binaire de n.
   Par exemple, l'appel **DecVersBin(10)** doit retourner [1, 0, 1, 0]
2. Ecrire une fonction **BinVersDec** prenant comme argument une liste Lc de bits et retournant l'entier naturel n dont l'écriture binaire est constituée de la liste Lc.
   Par exemple, l'appel **BinVersDec([1, 0, 1, 1])** doit retourner 11
3. Calculer la complexité de ces deux algorithmes en fonction de l'entier n soit donné en entrée soit retourné en sortie...

Exercice 2 : Exponentiation

On supposera que l'on dispose d'une fonction **DecBinLog** qui donne la décomposition binaire d'un entier n sous forme d'une liste et de complexité $\bigcirc(\log_2(n))$ Par exemple, **DecBinLog(10)** retourne [1, 0, 1, 0]

1. Exponentation naïve.
   (a) Ecrire une fonction **puissance** prenant comme argument un entier naturel n et un flottant r et retournant r^n en utilisant une boucle : *for k in range(n)* et sans utiliser l'opération **.
   (b) Vérifier que l'algorithme est correct et calculer sa complexité
2. Exponentation rapide On considère le programme suivant :

```
>>> def ExpoRap(n, r):
>>>         t = DecBinLog(n)
>>>         p, res = len(t), 1
>>>         for i in range(p):
>>>                 if t[i] == 1:
>>>                         res = res * res * r
>>>                 else :
>>>                         res = res * res
>>>         return(res)
```

   (a) Donner l'évolution des différentes variables locales utilisées par la fonction lors de l'appel : **ExpoRap(13, 2)**
   (b) Vérifier au moyen d'un invariant de boucle que l'appel à **ExpoRap(n, r)** donne bien r^n
   (c) Calculer la complexité de ce programme et vérifier qu'il porte bien son nom

Exercice 3 : Calcul de complexité sur la suite de Fibonacci

On définit la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ par : $F_0 = 0$, $F_1 = 1$ et la relation de récurrence :
$\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$.

1. Ecrire un programme itératif Python permettant de calculer F_n avec une complexité de $\bigcirc(n)$
2. On considère le programme récursif suivant

```
>>> def Fibo_Rec1(n):
>>>         if n <= 1 :
>>>                 return(n)
>>>         else :
>>>                 return (Fibo_Rec1(n-1) + Fibo_Rec1(n-2))
```

On note u_n le nombre d'additions nécessaires pour calculer **Fibo_Rec1(n)**.

(a) Déterminer une relation liant u_n, u_{n-1} et u_{n-2}

(b) En déduire que la suite $(v_n)_{n\in\mathbb{N}} = (1+u_n)_{n\in\mathbb{N}}$ vérifie une relation de récurrence linéaire double. Exprimer alors v_n en fonction de n

(c) En déduire la complexité de cet algorithme.

3. On considère un autre programme récursif

```
>>> def Fibo_Rec2(n, u, v):
>>>         if n == 0 :
>>>                 return(u)
>>>         else :
>>>                 return (Fibo_Rec2(n-1, v, u + v))
```

Le calcul de F_n se faisant par l'appel : **Fibo_Rec2(n, 0, 1)** On note u_n le nombre d'opérations élémentaires (additions, affectations, tests ...) nécessaires pour calculer **Fibo_Rec2(n, 0, 1)**.

(a) Déterminer une relation liant u_n et u_{n-1}.

(b) Exprimer alors u_n en fonction de n

(c) En déduire la complexité de cet algorithme.

On supposera par la suite que l'on dispose d'une fonction **DecBinLog** qui donne la décomposition binaire d'un entier n sous forme d'une liste et de complexité $\bigcirc(\log_2(n))$ Par exemple, DecBinLog(10) retourne [1, 0, 1, 0]

4. on peut montrer (et cela a en partie été fait dans le DL4) que l'on a :
$\forall p \in \mathbb{N}, F_{2p} = 2F_{p+1}F_p - F_p^2 \quad , \quad F_{2p+1} = F_{p+1}^2 + F_p^2 \quad \text{et} \quad F_{2p+2} = 2F_{p+1}F_p + F_{p+1}^2$
On considère alors le programme suivant

```
>>> def Fibo2(n):
>>>         t = DecBinLog(n)
>>>         p, u, v = len(t), 0, 1
>>>         for i in range(p):
>>>                 if t[i] == 0:
>>>                         u , v = 2 * u  * v - u * u , u * u + v * v
>>>                 else :
>>>                         u , v =  u  * u + v * v , 2 * u * v + v * v
>>>         return(u)
```

(a) Montrer, à l'aide d'un invariant de boucle par exemple, que l'appel à **Fibo2(n)** retourne bien F_n

(b) Calculer le nombre d'opérations élémentaires effectuées lors de l'appel à **Fibo2(n)** en fonction de la longueur p du tableau t.

(c) En déduire la complexité de ce programme.

Exercice 4 : Parties d'un ensemble

On considère l'ensemble $\mathbb{N}_n$ des entiers de 0 à n où n est un entier fixé.

1. Préambule Ecrire une fonction **estdedans** qui prend comme argument une liste L et un entier a et renvoie True si a est dans L et False sinon.
2. Représentation par une liste d'entiers On représente un sous-ensemble de $\mathbb{N}_n$ par une liste L d'entiers ordonnée. Par exemple le sous-ensemble $\{1, 2, 3, 5\}$ est représenté par la liste $[1, 2, 3, 5]$
   On considère la fonction suivante :

```
>>> def fonct1(L1, L2):
>>>         L = []
>>>         for k in range(n+1):
>>>                 if (estdedans(L1, k) and estdedans(L2, k)):
>>>                         L.append(k)
>>>         return(L)
```

   (a) Donner l'évolution des différentes variables locales utilisées par la fonction lors de l'appel : **fonct1([0, 1, 5], [1, 2])** avec l'hypothèse d'une variable globale n valant 5

   (b) Que fait **fonct1** ?

   (c) Écrire une fonction **complementaire** prenant comme argument une liste d'entiers L1 représentant un sous-ensemble de $\mathbb{N}_n$ et retournant la liste représentant le complémentaire de ce sous-ensemble.

   (d) Écrire une fonction **union** prenant comme argument deux listes d'entiers L1 et L2 représentant deux sous-ensembles de $\mathbb{N}_n$ et retournant la liste représentant la réunion de ces sous-ensembles

3. Représentation binaire On décide maintenant de représenter les sous-ensembles de $\mathbb{N}_n$ par une liste L de 0 et de 1. Si E est un sous-ensemble de $\mathbb{N}_n$, sa représentation est une liste L_E de longueur $n+1$: l'élément d'indice i de L_E vaut 1 si i est dans E et 0 sinon. Par exemple le sous-ensemble $\{1, 2, 3, 5\}$ de $\mathbb{N}_6$ est représenté par la liste $[0, 1, 1, 1, 0, 1, 0]$

   (a) Écrire une fonction **complementairebin** prenant comme argument une liste de bits L1 représentant un sous-ensemble de $\mathbb{N}_n$ et retournant la liste représentant le complémentaire de ce sous-ensemble.

   (b) Écrire une fonction **intersectionbin** prenant comme argument deux listes de bits L1 et L2 représentant deux sous-ensembles de $\mathbb{N}_n$ et retournant la liste représentant l'intersection de ces sous-ensembles

   (c) Écrire une fonction **unionbin** prenant comme argument deux listes de bits L1 et L2 représentant deux sous-ensembles de $\mathbb{N}_n$ et retournant la liste représentant la réunion de ces sous-ensembles

CORRECTION

Exercice 1 : Décomposition binaire

1.
```
>>> def DecVersBin(n):
>>>         m , t = n , [ ]
>>>         while m > 0 :
>>>                 t = [ m%2] + t
>>>                 m = m//2
>>>         return(t)
```

2.
```
>>> def BinVersDec(Lc):
>>>         m , p = Lc[0] , len(Lc)
>>>         for k in range(1,p) :
>>>                 m = 2*m + Lc[k]
>>>         return(m)
```

3. Si p est la longueur de la liste représentant n, on a : $2^{p-1} \leqslant n < 2^p$. On a donc $p = \left\lfloor \frac{\log(n)}{\log(2)} \right\rfloor + 1$.
Mais :
☞ Pour **DecVersBin**, on a $2p+2$ affectations, $2p$ calculs, p tests et p concaténations
☞ Pour **BinVersDec**, on a $p+1$ affectations, $p-1$ additions et $p-1$ multiplications.
Ainsi les deux fonctions ont **une complexité en $\bigcirc(\log(n))$**

Exercice 2 : Exponentiation

On supposera que l'on dispose d'une fonction **DecBinLog** qui donne la décomposition binaire d'un entier n sous forme d'une liste et de complexité $\bigcirc(\log_2(n))$

1. Exponentation naïve.

(a)
```
>>> def puissance(n ,r) :
>>>         res = 1
>>>         for k in range(n):
>>>                 res = res * r
>>>         return(res)
```

(b) Soit $u_k = \frac{res}{r^{k+1}}$ où res et k sont les variables locales intervenant dans la fonction puissance à la fin de chaque itération. Montrons qu'il s'agit d'un invariant de boucle. En effet entre l'étape k et l'étape $k+1$, on multiplie le dénominateur et le numérateur par r. Donc $u_{k+1} = u_k$. Or $u_0 = 1$ et la sortie de la boucle est réalisée avec $k = n-1$. Donc en sortie de boucle, res possède la valeur r^n. **L'algorithme est correct**
À chaque itération on effectue une multiplication et une affectation, donc
la complexité de l'algorithme est $\bigcirc(n)$

2. Exponentation rapide On considère le programme suivant :
```
>>> def ExpoRap(n, r):
>>>         t = DecBinLog(n)
>>>         p, res = len(t), 1
>>>         for i in range(p):
>>>                 if t[i] == 1:
>>>                         res = res * res * r
>>>                 else :
>>>                         res = res * res
>>>         return(res)
```

(a) État des variables pour l'appel : **ExpoRap(13, 2)**.

instruction	t	p	res en début de boucle	i	t[i]	res à la fin
t = DecBinLog(13)	[1,1,0,1]					
p, res = len(t), 1	[1,1,0,1]	4	1			
boucle	[1,1,0,1]	4	1			
i = 0	[1,1,0,1]	4	1	0	1	2
i = 1	[1,1,0,1]	4	2	1	1	8
i = 2	[1,1,0,1]	4	8	2	0	64
i = 3	[1,1,0,1]	4	64	3	1	8192
sortie de boucle	[1,1,0,1]	4	8192	3	1	8192

(b) On remarque d'abord que l'entier n vaut $n = \sum_{k=0}^{p-1} t[k]2^{p-1-k}$

Soit $u_i = \dfrac{r^{2^{i+1}n//2^p}}{res}$ où res, p et i sont les variables locales intervenant dans la fonction ExpoRap(n, r) en fin d'itération. Montrons qu'il s'agit d'un invariant de boucle.
En effet entre l'étape i et l'étape $i+1$, on multiplie la puissance de r par 2 et on ajoute 0 ou 1 selon que le bit corrrespondant à 2^{p-i-2} vaut 0 ou 1 i.e. selon que t[i+1] vaut 0 ou 1. Donc le numérateur est élevé au carré et est multiplié par 1 ou r selon que t[i+1] vaut 0 ou 1, ce qui est exactement le sort que subit la variable res. Donc $u_{i+1} = u_i$. En entrée de boucle, on a $u_0 = 1$ donc en sortie on a aussi $u_{p-1} = 1$ i.e. res vaut r^n en sortie de boucle.
L'appel à ExpoRap(n, r) donne bien r^n

(c) À chaque itération, on a une affectation et 1 ou 2 multiplications. Donc au total on a entre $2p+3$ et $3p+3$ opérations élémentaires ainsi que le nombre de calculs utilisés par **DecBinLog(n)** qui est en $\bigcirc(\log_2(n))$. Comme $p = \left\lfloor \dfrac{\log(n)}{\log(2)} \right\rfloor + 1$, on en déduit que
ExpoRap(n ,r) a une complexité en $\bigcirc(\log_2(n))$ ce qui est bien plus rapide que l'exponentiation naïve.

Exercice 3 : Calcul de complexité sur la suite de Fibonacci

On définit la suite de Fibonacci $(F_n)_{n\in\mathbb{N}}$ par : $F_0 = 0$, $F_1 = 1$ et la relation de récurrence :
$\forall n \in \mathbb{N}, F_{n+2} = F_{n+1} + F_n$.

1. Ecrire un programme itératif Python permettant de calculer F_n avec une complexité de $\bigcirc(n)$

```
>>> def fibo(n):
>>>         u, v = 0, 1
>>>         for k in range(n) :
>>>                 w = v
>>>                 v = u + v
>>>                 u = w
>>>         return(u)
```

2. On considère le programme récursif suivant

```
>>> def Fibo_Rec1(n):
>>>         if n <= 1 :
>>>                 return(n)
>>>         else :
>>>                 return (Fibo_Rec1(n-1) + Fibo_Rec1(n-2))
```

On note u_n le nombre d'additions nécessaires pour calculer **Fibo_Rec1(n)**.

(a) Le calcul de Fibo_Rec1(n) nécessite 1 addition et les calculs de Fibo_Rec1(n-1) et Fibo_Rec1(n-2). Ainsi $\boxed{\forall n \geqslant 2, u_n = u_{n-1} + u_{n-2} + 1}$

(b) on en déduit : $\forall n \geqslant 2, v_n = v_{n-1} + v_{n-2}$ avec $v_0 = v_1 = 1$. Or l'ensemble des suites vérifiant cette relation est un espace-vectoriel de dimension 2 dont une base est constituée des suites $(\phi^n)_{n\in\mathbb{N}}$ et $(\beta^n)_{n\in\mathbb{N}}$ avec $\phi = \dfrac{1+\sqrt{5}}{2}$ et $\beta = \dfrac{1-\sqrt{5}}{2}$. Après simplification, on trouve : $\boxed{\forall n \in \mathbb{N}, v_n = \dfrac{\phi^{n+1} - \beta^{n+1}}{\sqrt{5}}}$

(c) Comme u_n et v_n ont le même ordre de grandeur et que $|\beta| < 1 < \phi$, on en déduit que $\boxed{\textbf{la complexité de cet algorithme est } \bigcirc(\phi^n)}$

3. On considère un autre programme récursif

```
>>> def Fibo_Rec2(n, u, v):
>>>         if n == 0 :
>>>                 return(u)
>>>         else :
>>>                 return (Fibo_Rec2(n-1, v, u + v))
```

Le calcul de F_n se faisant par l'appel : **Fibo_Rec2(n, 0, 1)** On note u_n le nombre d'opérations élémentaires (additions, affectations, tests ...) nécessaires pour calculer **Fibo_Rec2(n, 0, 1)**.

(a) Le calcul de Fibo_Rec2(n, u, v) nécessite un test et deux additions de plus que le calcul de Fibo_Rec2(n-1, u', v'). Donc on a : $\boxed{\forall n \in \mathbb{N}^*, u_n = u_{n-1} + 3}$

(b) On en déduit que $\boxed{\forall n \in \mathbb{N}^*, u_n = 3n + 1}$

(c) $\boxed{\textbf{La complexité de cet algorithme est donc } \bigcirc(n)}$.

On supposera par la suite que l'on dispose d'une fonction **DecBinLog** qui donne la décomposition binaire d'un entier n sous forme d'une liste et de complexité $\bigcirc(\log_2(n))$

4. on peut montrer (et cela a en partie été fait dans le DL4) que l'on a :
$\forall p \in \mathbb{N}, F_{2p} = 2F_{p+1}F_p - F_p^2 \quad , \quad F_{2p+1} = F_{p+1}^2 + F_p^2 \quad \text{et} \quad F_{2p+2} = 2F_{p+1}F_p + F_{p+1}^2$
On considère alors le programme suivant

```
>>> def Fibo2(n):
>>>         t = DecBinLog(n)
>>>         p, u, v = len(t), 0, 1
>>>         for i in range(p):
>>>                 if t[i] == 0:
>>>                         u , v = 2 * u  * v - u * u , u * u + v * v
>>>                 else :
>>>                         u , v =  u  * u + v * v , 2 * u * v + v * v
>>>         return(u)
```

(a) On remarque d'abord que l'entier n vaut $n = \sum_{k=0}^{p-1} t[k]2^{p-1-k}$. Montrons qu'à la fin de chaque itération, la variable u contient la valeur F_{q_i} et v la valeur F_{q_i+1} où $q_i = \sum_{k=0}^{i} t[k]2^{i-k}$.

A la fin de l'itération $i = 0$, u prend la valeur $1 = F_1$, v la valeur $1 = F_2$ alors que $q_0 = t[0] = 1$
Si on a bien les états des variables proposés en fin d'itération i. A l'itération i+1.
- q_{i+1} vaut $2q_i + t[i+1]$ i.e. $2q_i$ si $t[i+1]$ vaut 0 et $2q_i + 1$ sinon.
- si t[i+1] vaut 0 En fin d'itération, u vaut $2F_{q_i}F_{q_i+1} - F_{q_i}^2 = F_{2q_i}$ et v vaut $F_{q_i+1}^2 + F_{q_i}^2 = F_{2q_i+1}$

– si t[i+1] vaut 1 En fin d'itération, u vaut $F_{q_i+1}^2 + F_{q_i}^2 = F_{2q_i+1}$ et v vaut $2F_{q_i}F_{q_i+1} + F_{q_i+1}^2 = F_{2q_i+2}$

Ainsi dans tous les cas, en fin d'itération i+1, u contient la valeur $F_{q_{i+1}}$ et v la valeur $F_{q_{i+1}+1}$

On en déduit donc que **l'appel à Fibo2(n) retourne bien F_n**

(b) À chaque itération, on effectue un test, 5 multiplications, 2 additions et 2 affectations. Donc il y a $8p+5$ opérations élémentaires en plus des calculs utilisés par **DecBinLog(n)** qui sont en $\bigcirc(\log_2(n))$

(c) **Fibo2(n) a donc une complexité en $\bigcirc(\log_2(n))$**

Exercice 4 : Parties d'un ensemble

On considère l'ensemble $\mathbb{N}_n$ des entiers de 0 à n où n est un entier fixé.

1. Préambule

```
>>> def estdedans1(L, a):
>>>         for k in range(len(L)):
>>>                 if L[k] == a:
>>>
    return(True)
>>>         return(False)
```

Autre version

```
>>> def estdedans2(L, a):
>>>         return(a in L)
```

2. Représentation par une liste d'entiers On représente un sous-ensemble de $\mathbb{N}_n$ par une liste L d'entiers ordonnée.

On considère la fonction suivante :

```
>>> def fonct1(L1, L2):
>>>         L = []
>>>         for k in range(n+1):
>>>                 if (estdedans(L1, k) and estdedans(L2, k)):
>>>                         L.append(k)
>>>         return(L)
```

(a) Évolution des différentes variables locales lors de l'appel : **fonct1([0, 1, 5], [1, 2])**

instruction	L en début de boucle	k	k dans L1	k dans L2	Là la fin
L = [ ]	[ ]				[ ]
boucle	[ ]				[ ]
k = 0	[ ]	0	True	False	[ ]
k = 1	[ ]	1	True	True	[1]
k = 2	[1]	2	False	True	[1]
k = 3	[1]	3	False	False	[1]
k = 4	[1]	4	False	False	[1]
k = 5	[1]	5	True	False	[1]
sortie de boucle	[1]	5	True	False	[1]

(b) **fonct1 réalise l'intersection des deux sous-ensembles**

(c)

```
>>> def complementaire(L1):
>>>         L = []
>>>         for k in range(n+1):
```

```
>>>                 if not(estdedans(L1, k)):
>>>                         L.append(k)
>>>         return(L)
```

(d)

```
>>> def union(L1, L2):
>>>         L = []
>>>         for k in range(n+1):
>>>                 if (estdedans(L1, k) or estdedans(L2, k)):
>>>                         L.append(k)
>>>         return(L)
```

3. Représentation binaire On décide maintenant de représenter les sous-ensembles de $\mathbb{N}_n$ par une liste L de 0 et de 1. Si E est un sous-ensemble de $\mathbb{N}_n$, sa représentation est une liste L_E de longueur $n+1$: l'élément d'indice i de L_E vaut 1 si i est dans E et 0 sinon. Par exemple le sous-ensemble $\{1,2,3,5\}$ de $\mathbb{N}_6$ est représenté par la liste $[0,1,1,1,0,1,0]$

(a)

```
>>> def complementairebin(L1):
>>>         L = []
>>>         for k in range(n+1):
>>>
    L.append(1-L1[k])
>>>         return(L)
```

(b)

```
>>> def intersectionbin(L1, L2):
>>>         L = []
>>>         for k in range(n+1):
>>>                 L.append(L1[k]
    * L2[k])
>>>         return(L)
```

(c)

```
>>> def unionbin(L1, L2):
>>>         L = []
>>>         for k in range(n+1):
>>>                 if L1[k] +
    L2[k] >= 1:
>>>
    L.append(1)
>>>                 else :
>>>
    L.append(0)
>>>         return(L)
```