Concours Blanc d'Informatique – Corrigé

/ 3 **1)**

```
def cardinal(L):
    s=0
    p=len(L)
    for i in range(p):
        if L[i]==True: # i est dans X ssi L[i] vaut True
            s=s+1
    return s
```

/ 4 **2)**

```
def rajout(L,m):
    if m>=len(L):
        while len(L)<m: # on ajoute des False jusqu'à ce que L
            soit de longueur m
            L.append(False)    # version avec effet de bord
            # ou L = L+[False] : version sans effet de bord
        return L
    else:
        print("ErReUr FaTaLe") # print pour afficher
        return None
```

/ 2 **3) a)** Sans les lignes 3 et 4, si `L1` et `L2` n'ont pas la même longueur, alors l'une est de longueur `m` et l'autre de longueur $< m$. Ainsi, l'instruction `L1[i]` ou `L2[i]` renverrait une erreur de dépassement d'indice ("out of bonds") lorsque `i` est égal à `m-1` (entre autres).

/ 3,5 **b)** *(La question présuppose aussi qu'on a compilé le code de la fonction `rajout`)*
`M=[True, False, True, True]` (ou `M=[True,False,True]` si rajout n'a pas d'effet de bord)
`N=[False, False, True, True]`
`P=[True, False, True, True]`
`L1` et `L2` renverraient une erreur, car ces variables ne sont pas définies (elles ne sont que des variables locales dans la fonction `union`)

/ 5 **c)** Si `rajout` possède un effet de bord, alors `union` aussi. Sinon, alors `union` n'en possède pas. La fonction `union2` de gauche supprime tout effet de bord. La fonction `union2` de droite ajoute un effet de bord avec un `.append` :

```
def union2(L1,L2):
    M1 = L1.copy()
    M2 = L2.copy()
    m = max(len(M1),len(M2))
    M1 = rajout(M1,m)
    M2 = rajout(M2,m)
    for i in range(m):
        if M2[i]==True:
            M1[i]=True
    return M1
```

```
def union2(L1,L2):
    m = max(len(L1),len(L2))
    for i in range(m):
        L1.append(False)
        L2.append(False)
        if L2[i]==True:
            L1[i]=True
    return L1
# L1 peut être de taille > m
# mais ça ne pose pas de pb
```

/ 5

4)

```
1  def diff(L1,L2):                # version sans effet de bord
2      m = max(len(L1),len(L2))
3      L1 = rajout(L1,m)
4      L2 = rajout(L2,m)
5      M = L1.copy()               # M sera la booliste de X1\X2
6      for i in range(m):
7          if L2[i]==True:         # si i est dans X2
8              M[i]=False          # il ne doit pas être dans X1\X2
9      return M
```

/ 1,5

5) $75 = \overline{1001011}^{(2)}$, on voit qu'il y a un 1 à la position i si et seulement si $6 - i$ est dans l'ensemble $\{0, 1, 3, 6\}$.

/ 3

6) $13 = \overline{1101}^{(2)}$ et donc 13 est représenté par l'ensemble $\{0, 1, 3\}$.
$151 = \overline{10010111}^{(2)}$ et donc 151 est représenté par l'ensemble $\{0, 1, 2, 4, 7\}$.

/ 4

7) On remarque que si $X \subset Y$ alors $s_X \leq s_Y$ (avec s_X et s_Y les entiers représentant X et Y). Alors, le plus grand entier possible sera celui qui représente E, donc

$$\sum_{i \in E} 2^i = 2^0 + 2^1 + \ldots + 2^{n-1}$$
$$= \frac{2^n - 1}{2 - 1} = \boxed{2^n - 1}$$

/ 4

8)

```
1  def set2int(L):
2      s = 0
3      m = len(L)
4      for i in range(m):
5          if L[i]==True:
6              s = s + 2**i
7      return s
```

/ 4

9) Il y a 3 opérations élémentaires : `+` et `==` en ligne 5 et $**$ en ligne 6 (on peut éventuellement compter `len(L)` ligne 3 mais cela ne change rien au résultat final).

Le pire cas se produit lorsque `L` ne contient que des `True`, auquel cas chaque opération est répétée m fois. Il y a donc $3m$ opérations élémentaires dans le pire cas.

La complexité est donc $\boxed{\text{d'ordre } m}$, ou encore linéaire.

/ 11

10)

```
def int2set(s):
    if type(s)==int and s>=0:
        L = [] % la booliste à retourner
        # si s=0, on ne fait rien et on retournera []
        while s>0:
            # sinon, on fait comme pour calculer la décomposition
               en base 2, avec des divisions euclidiennes
               successives. Mais au lieu de stocker le reste, on
               stocke True si le reste vaut 1, et False sinon.
            if s%2 == 1:
                L.append(True)
            else:
                L.append(False)
            s = s//2
            # à présent, L[0] vaut True si il y a 1*2**0 dans
               l'écriture de p en base 2 et False si c'est 0*2**0,
               etc.
        return L
    else:
        print("erreur, s n'est pas un entier naturel")
        return None
```

/ 4

11) On affirme que $\sigma_n = 2^n - 1$.
(Cette question présuppose que les sous-ensembles de $\mathcal{G}$ soient disjoints, sinon des contre-exemples existent : par exemple avec $n = 4$ donc $E = \{0, 1, 2, 3\}$, alors $\sigma_n = 15$ peut être atteint avec $1+2+3+4+5$, qui représentent les ensembles $\{0\}$, $\{1\}$, $\{0, 1\}$, $\{2\}$ et $\{0, 2\}$ mais aucun de ces sous-ensembles ne contient 3.)

En effet, en supposant les sous-ensembles de $\mathcal{G}$ disjoints, si le point i est pris une (et une seule) fois, alors il engendre une contribution à hauteur de 2^i dans la somme en question. Le total sera donc $\sum_{i=0}^{n-1} 2^i = \boxed{2^n - 1}$.

/ 6

12)

```
def sansDoublon(L):
    m = len(L)
    for k in range(m):
        if L[k] in L[0:k]: # si L[k] est présent dans les valeurs
           des indices 0 à k-1
            return False
    return True # si on n'a jamais retourné False, alors il n'y a
       pas de doublon
```

/ 8 **13)** La fonction `sort` (ou équivalent) permet de trier immédiatement par ordre croissant et d'avoir quelques points (il reste ensuite à inverser l'ordre), mais pas la totalité.

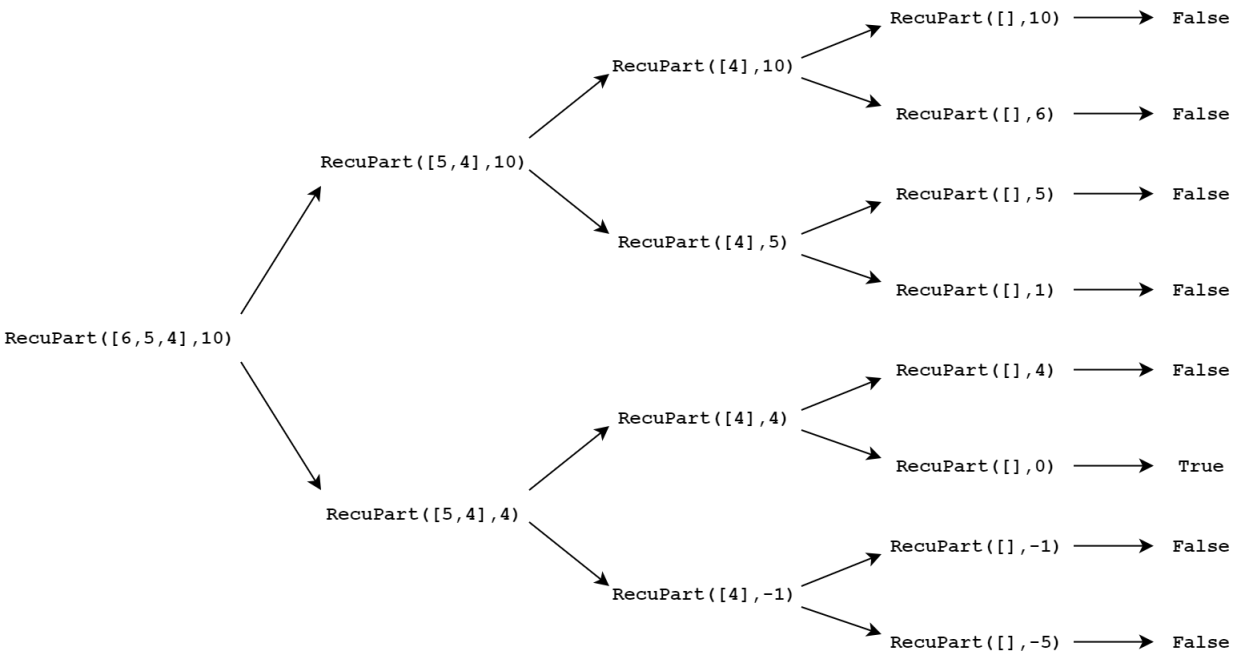
```
def tri(L): # tri par insertion décroissant (par exemple)
    m = len(L)
    for k in range(m):
        # on cherche l'indice (noté ind) du plus grand élément de
           L[k:], dont la valeur sera max
        max = L[k]
        ind = k
        for j in range(k+1,m):
            if L[j]>max:
                max = L[j]
                ind = j
        # on échange les éléments des indices k et ind
        L[ind],L[k]=L[k],L[ind]
    return L
```

/ 6 **14)**

```
def glouton(L,sum):
    L = tri(L)
    M = []
    sommeM = 0 # somme des éléments de m
    m = len(L)
    for i in range(m):
        if sommeM + L[i] <= sum:
            M.append(L[i])
            sommeM = sommeM + L[i]
    if sum == 2**m-1:
        return M
    else:
        return None
```

/ 4 **15)** Ce n'est pas toujours le cas, en effet avec `L = [10,2,5]` et `sum = 15`, alors l'algorithme glouton trouvera `M = [10,2]` alors que la solution est `[10,5]`.

/ 2,5 **16)** Cf page suivante.



/ 5,5

17) Montrons par récurrence sur la longueur de `L` que l'instruction `recuPart(L,sum)` termine (quelle que soit la valeur de `sum`).

- Si `L` est de longueur 0, alors l'instruction termine par les lignes 7-8 (voire les lignes 5-6 si `sum` vaut 0)
- Soit $p \in \mathbb{N}$. Supposons par récurrence que `recuPart(L,sum)` termine lorsque `L` a pour longueur p. Montrons que c'est encore le cas si `L` a pour longueur $p+1$.

  Si `L` a pour longueur $p+1$, alors ou bien `sum` vaut 0 et l'algorithme termine, ou bien `sum` est non nul et aucune des deux conditions d'arrêt ne sont vérifiées (car $p+1 > 0$). Alors, l'algorithme exécute les instructions des lignes 14 et 17, qui réalisent des appels de la forme `recuPart(L[1:],...)`. Par hypothèse de récurrence, ces appels terminent car `L[1:]` est une liste de taille p. Donc, on atteint après un nombre fini d'opérations la ligne 20, qui termine également. Ainsi, l'algorithme termine pour le rang $p+1$.
- Finalement, l'instruction `recuPart(L,sum)` termine toujours quelle que soit la longueur de `L` et la valeur de `sum`.

/ 3

18) Calculons c_n avec $n \in \mathbb{N}$. Les opérations élémentaires sont les deux `==` aux lignes 5 et 7, ainsi que le `-` ligne 17 et le `or` ligne 20. Si $n \geq 1$, il y a également deux appels récursifs aux lignes 14 et 17 avec des listes de tailles $n-1$, qui comptent donc pour $2c_{n-1}$ opérations élémentaires.

Lorsqu'on est dans le pire cas, la condition d'arrêt ligne 5 n'est jamais vérifiée (pour aucun des appels récursifs). Si $n \geq 1$, la condition d'arrêt ligne 7 n'est pas vérifiée si bien que toutes les opérations élémentaires listées ci-dessus sont réalisées. On a donc

$$\forall n \geq 1 \qquad \boxed{c_n = 2c_{n-1} + 4}$$

/ 4

19) Il suffit de calculer c_n. On remarque que (pour tout $n \geq 1$)

$$c_n + 4 = 2(c_{n-1} + 4)$$

si bien que $(c_n + 4)$ est une suite géométrique de raison 2. On a ainsi

$$c_n + 4 = 2^n(c_0 + 4)$$

Or, dans le pire cas on a $c_0 = 2$ (l'appel se termine ligne 8), et donc

$$c_n = 6 \times 2^n - 4$$

On en déduit que la complexité est exponentielle. Cette complexité est trop grande pour être exploitable pour de grandes valeurs de n (pour $n = 10^6$, on a $c_n \approx 2^{10^6} = (2^{10})^6 \approx 1000^6 = 10^{18}$ soit à peu près un milliard de milliards d'opérations élémentaires).

/ 4

20) On considère l'ensemble $E = \{0, 1, 2, 3, 4, 5\}$. On dispose d'un contre-exemple avec la famille

$$\mathcal{F} = (X_0, X_1, X_2) := (\{0, 2, 4\}, \{1, 3, 5\}, \{0, 1, 2, 3\})$$

L'algorithme glouton prendrait comme premier ensemble $X_2 = \{0, 1, 2, 3\}$ qui couvre 4 éléments, soit plus que X_0 et X_1, puis devrait rajouter X_0 pour couvrir 4 puis X_1 pour couvrir 5. Ainsi, il retournerait $\mathcal{F}$ toute entière.
Toutefois $\mathcal{F}$ n'est pas une famille couvrante optimale puisque (X_1, X_2) est également couvrante et utilise moins d'ensembles.

/ 4

21) (Il y avait une erreur d'énoncé ici : les ensembles S_1 et S_2 sont à remplacer par X_1 et X_2)

```
1  def reste(s1,s2):
2      return cardinal(diff(s1,s2))
```

(L'énoncé indique en fin de partie II que les fonctions de la partie I peuvent s'utiliser en prenant comme arguments des entiers qui représentent les ensembles plutôt que des boolistes)

/ 12

22)

```
1  def glouton():
2      # F et n sont accessibles sans les passer en argument
3
4      r = len(F) # nombre de sous-ensembles de F
5      h = 0 # entier qui représente la sous-famille H de F
6      NC = 2**n-1 # entier qui représente l'ensemble des éléments
           non couverts (initialement il représente E)
7
8      while NC!=0 :  # tant qu'on n'a pas tout couvert
9
10         i = 0  # indice du sous-ensemble de F qui va couvrir le
               plus d'éléments restants
11         for k in range(r):
12             if reste(NC,F[i]) < reste(NC,F[k]): # si k couvre plus
                   d'indices que i
13                 i=k  # i deviendra ce k
14
15         # on enlève des éléments non-couverts ceux qui sont
               couverts par F[i]
16         NC = diff(NC,F[i])
17         h = h + 2**i # on ajoute 2**i à h pour signifier qu'on
               prend le sous-ensemble d'indice i de F
18     return h
```

/ 12 **23)**

```python
def couverture(h):
    C = 0 # ensemble des éléments couverts
    L = int2set(h) # booliste : L[k] vaut True si l'ensemble F[k]
        est dans la famille H ou non.
    m = len(L)
    for k in range(m):
        if L[k]==True:
            C = union(C,F[k]) # on fait ainsi l'union de tous les
                F[k] qui sont dans la famille H

    if C==2**n-1: # on a tout couvert
        return True
    else:
        return False
```

/ 16 **24)**

```python
def optimale():
    r = len(F)
    hmax = 2**r # chaque entier entre 0 et hmax représente une
        sous-famille H possible de F

    hfinal = hmax # entier qui représente la sous-famille qu'on va
        renvoyer (initialement tout F)
    NSEfinal = r # nombre de sous-ensembles de cette famille
        hfinal (initialement tous ceux de F donc r)

    for h in range(hmax):
        if couverture(h)==True: # si h est couvrante
            NSE = cardinal(h) # nombre de sous-ensembles de h
            if NSE < NSEfinal: # si on couvre avec moins
                d'ensembles que hfinal
                hfinal = h
                NSEfinal = cardinal(h)
    return hfinal
```