Correction DS 3 d'informatique

I Représentation par des listes

1. (/0.5)

```
def card(L):
    return len(L)
```

2. (/1)

```
def appartient(k, L):
    for i in range(len(L)):
        if L[i] == k:
            return True
    return False
```

Attention à bien mettre le `return False` après la boucle for, et pas dedans.

3. (a) (/1.5)

```
def min(L):
    m = L[0]
    for i in range(1, len(L)):
        if L[i] < m:
            m = L[i]
    return m
```

(b) (/1) On montre, $\forall k \in \mathbb{N}^*$, H_k : au début de la boucle for pour $i = k$, `m` est le minimum de `L[0]`, ..., `L[k-1]`.
Initialisation et induction sont évidentes.

(c) (/0.5) Dans le pire des cas la boucle for s'exécute `len(L)-1` fois et chaque itération réalise 2 opérations élémentaires (`L[i] < m` et `m = L[i]`). En ajoutant les opérations `m = L[0]` et `len(L)`, on obtient donc une complexité dans le pire des cas: 2 (`len(L)`-1) + 2 = 2 `len(L)` = $O(n)$.

(d) (/0.5) Cet ordinateur réalise 2×10^9 opérations à la seconde, et doit réaliser 2×10^6 opérations. Il met donc un temps estimé de $\frac{2 \times 10^6}{2 \times 10^9} = 1$ ms.

4. (a) (/2)

```
def est_ens(L):
    for i in range(len(L)):
        for j in range(i+1, len(L)):
            if L[i] == L[j]:
                return False
    return True
```

Il fallait s'assurer que `i` $\neq$ `j` quand on compare `L[i]` et `L[j]`.
(On pouvait aussir réutiliser `appartient`:

```
def est_ens(L):
    for i in range(len(L)):
        if appartient(L[i], L[i+1:]):
            return False
    return True
```

)

(b) (/0.5) La boucle `for i in range(len(L))` s'exécute `n` fois et, à chaque fois, la boucle `for j in range(i+1, len(L))` s'exécute au plus `n` fois. La complexité dans le pire des cas est donc $O(n^2)$.

(c) (/0.5) Estimation: $\frac{10^{12}}{2 \times 10^9} = 500$s.

5. (a) (/2)

```
def inter(L1, L2):
    res = []
    for i in range(len(L1)):
        if appartient(L1[i], L2):
            res.append(L1[i])
    return res
```

(b) (/0.5) `appartient(L1[i], L2)` a une complexité $O(n2)$ et est exécuté $n1$ fois, ce qui donne une complexité totale dans le pire des cas de $O(n1 \times n2)$.

6. (/2)

```
def union(L1, L2):
    res = L1[:] # copie de L1
    for i in range(len(L2)):
        if not appartient(L2[i], res):
            res.append(L2[i])
    return res
```

En vérifiant que `L2[i]` n'appartient pas déjà à `res` on évite les doublons.
Il était possible d'utiliser `pop`, à condition de prendre en compte le décalage des indices qu'il engendre.

II Représentation binaire

1. (/0.5) $< \underbrace{0...0}_{n} >_2 = 0$.
2. (/1) $< \underbrace{1...1}_{n} >_2 = 2^n - 1$.
3. (/0.5) $< 0...01\underbrace{0...0}_{k} >_2 = 2^k$.
4. (/1) $26 = 2^4 + 2^3 + 2^1 = < 11010 >_2$ représente $\{1, 3, 4\}$.
5. (/0.5) a & b représente $A \cap B$ et $a \mid b$ représente $A \cup B$.
6. (/1)

```
def inter2(a, b):
    return a & b

def union2(a, b):
    return a | b
```

L'intersection et l'union se fait en temps constant (une seule opération élémentaire) au lieu d'un temps quadratique, ce qui est une énorme différence.

7. (/1)

```
def appartient2(k, a):
    return inter2(2**k, a) != 0
```

On a vu en 3. que $\{k\}$ est représenté par 2^k.

8. (/1.5)

```
def card2(a):
    res = 0
    for i in range(n):
        if appartient2(i, a):
            res += 1
    return res
```

Pour savoir quand arrêter la boucle for, on peut soit utiliser le fait que la longueur de a en

binaire est $\lfloor \log_2(a) + 1 \rfloor$, soit utiliser le n de l'énoncé.
On peut noter que `card2` est moins efficace que `card`.

9. (/1) $26 = < 011010 >_2 \rightsquigarrow < 100101 >_2 \rightsquigarrow < 100101 >_2 + 1 = < 100110 >_2$
10. (/1) 26 représente $\{1, 3, 4\}$ et 26 & -26 $= < 000010 >_2$ représente $\{1\}$.
11. (/1.5) On remarque que 26 & -26 représente le minimum de l'ensemble représenté par 26.

    De manière général, soit a un entier représentant un ensemble A. Supposons que m soit le minimum de A.

    Alors $a =< b_1...b_p1\underbrace{0...0}_{m} >_2$ où $b_1...b_p$ est une suite de bits (inconnue). -a est obtenu par complément à 2 (si b est un bit, $\overline{b}$ est le bit inversé):

    $$< b_1...b_p1\underbrace{0...0}_{m} >_2 \rightsquigarrow < \overline{b_1}...\overline{b_p}0\underbrace{1...1}_{m} >_2 \rightsquigarrow < \overline{b_1}...\overline{b_p}0\underbrace{1...1}_{m} >_2 + 1 = < \overline{b_1}...\overline{b_p}1\underbrace{0...0}_{m} >_2$$

    La dernière égalité étant obtenu à cause de la retenue.

    Donc a & -a $= < b_1...b_p1\underbrace{0...0}_{m} >_2$ & $< \overline{b_1}...\overline{b_p}1\underbrace{0...0}_{m} >_2 = < 0...01\underbrace{0...0}_{m} >_2 = 2^m$, et a & -a représente bien $\{m\}$.

    On en déduit:

```python
def min2(a):
    return a & -a
```

    Si l'on veut renvoyer m plutôt que la représentation de $\{m\}$, on peut prendre le logarithme en base 2 (`log2` du module `math`).
    Dans tous les cas, `min2` est beaucoup plus rapide que `min`.
    Remarque: il n'existe pas "formule magique" dans le même genre pour calculer le maximum d'un ensemble.

Compléments:

& et | sont des opérations directement applicable par le processeur: en langage **assembleur** (le "langage du processeur", très proche de la machine) il existe des fonctions qui les implémentent (dont le nom peut dépendre du processeur utilisé). Une autre opération intéressante, appelé XOR et noté ^ dans python, réalise le **ou exclusif** (le $i$ème bit de $a\hat{}b$ est égal à 1 si exactement un des bits de a ou b est égal à 1). Si a et b représente des ensembles, cela revient à faire la **différence symétrique** de ces ensembles. Ainsi, la différence symétrique se fait en une seule opération avec cette représentation.

Une autre utilité de la représentation binaire des ensembles réside dans le fait que l'on peut facilement **énumérer** les sous-ensembles de $\{0, ..., n-1\}$: il suffit de parcourir les entiers de 0 à $2^n - 1$. Cette tâche est beaucoup plus ardue avec les listes.
Par exemple, pour calculer le nombre de sous ensembles de taille pair de $\{0, ..., n-1\}$, on peut écrire:

```python
nb_pairs = 0
for i in range(2**n):
    if card2(i) % 2 == 0:
        nb_pairs += 1
```

(Dans ce cas particulier, on aurait aussi pu être plus intelligent et remarquer que[1] ce nombre est égal à 2^{n-1})

[1] Exercice laissé au lecteur: le prouver.