D.S. 1

Couverture optimale

D'après X PSI 2012

1 Autour des ensembles

Question 0

```
def parite(k):
    return k%2
```

Question 1

```
def cardinal(t):
    n = len(t)
    card  = 0
    for i in range(n):
        if t[i]:
            card = card + 1
    return card
```

Question 2
Attention : on se sait pas si i est un indice du tableau !

```
def appartient(i, t):
    n = len(t)
    if i < n:
        return t[i]
    else:
        return False
```

L'évaluation paresseuse permet d'écrire aussi

```
def appartient(i, t):
    return (i < len(t)) and t[i]
```

Mais est-ce lisible ?

Dans les deux questions suivantes t_1 et t_2 n'ont peut être pas la même longueur.

Question 3
Ici on peut copier le tableau t_1 et donner la valeur FAUX aux éléments de t_2.

```
def diff(t1, t2):
    n1 = len(t1)
    n2 = len(t2)
    dif = [False]*n1
    for i in range(n1):
        dif[i] = t1[i]
    for i in range(min(n1, n2)):
        if t2[i]:
            dif[i] = False
    return dif
```

Cependant la meilleure réponse est d'utiliser une question précédente.

```
def diff(t1, t2):
    n1 = len(t1)
    n2 = len(t2)
    n = max(n1, n2)
    dif = [False]*n
    for i in range(n):
        dif[i] = appartient(i, t1) and not appartient(i, t2)
    return dif
```

Question 4
L'algorithme est semblable, l'union a la taille maximale et ajoute les éléments de t_1 puis de t_2.

```
def union(t1, t2):
    n1 = len(t1)
    n2 = len(t2)
    uni = [False]*max(n1, n2)
    for i in range(n1):
        uni[i] = t1[i]
    for i in range(n2):
        if t2[i]:          # ou uni[i] = uni[i] or t2[i]
            uni[i] = True
    return dif
```

Ou on utilise ce qu'on a déjà écrit

```
def diff(t1, t2):
    n1 = len(t1)
    n2 = len(t2)
    n = max(n1, n2)
    uni = [False]*n
    for i in range(n):
        uni[i] = appartient(i, t1) or appartient(i, t2)
    return dif
```

2 Représentation par entiers

Question 5

U est un sous-ensemble associé à $\sum_{k=0}^{n-1} 2^k = 2^n - 1$ qui est la valeur maximale.

$\emptyset$ est un sous-ensemble associé à 0 qui est la valeur maximale.

Question 6

Pour garder une complexité linéaire on calcule la puissance de 2 par incrément à chaque étape de la boucle ; on fait donc n multiplications et au plus n additions si n est longueur de la liste.

```
def set2int(t):
    n = len(t)
    num = 0
    exp = 1
    for i in range(n):
        if t[i]:
            num = num + exp
        exp = exp*2
    return num
```

Question 7

On choisit de créer la représentation de longueur minimale

```
def int2set(n):
    ens = []
    while n > 0:
        ens.append(n%2 == 1)
        n = n//2
    return ens
```

La complexité est alors la taille de la liste en sortie qui est de l'ordre de $\log_2(n)$.

On n'a pas besoin des fonctions de conversions. Il existe des opérateurs qui travaillent bit par bit : `&` *qui calcule le le produit des bits et* `|` *qui calcule le maximum des bits.*
On peut aussi décaler les bits :
`n << p` *décale* n *de* p *zéros vers les poids forts (on multiplie par* 2^p*),*
`n >> p` *décale* n *de* p *zéros vers les poids faibles (on multiplie par* 2^p *dans* $\mathbb{N}$*).*
Seule la fonction cardinal demande une boucle.

```
def union(n1, n2):
    return n1 | n2

def diff(n1, n2):
    return (n1 | n2) - n2

def appartient(i, n):
    return (n & 1 << i) != 0

def cardinal(n):
    card = 0
    while n != 0:
        card = card + n%2
        n = n//2
    return card
```

3 Familles, sous-familles et couvertures

Question 8
U est l'ensemble des élèves.
Pour chaque activité on considère l'ensemble des élèves qui sont intéressés par celle-ci.
Comme chaque élève a choisi au moins une activité les ensembles définis ci-dessus couvrent U.
Chercher un ensemble minimal d'activité convenant à chaque élève revient alors à trouver une couverture optimale.

Question 9
Pour $U = \{0,1,2,3,4,5\}$ et $F = \big\{\{0,2,4\},\{0,1\},\{2,3\},\{4,5\}\big\}$, l'algorithme glouton choisira d'abord le sous-ensemble de cardinal 3 et devra ensuite ajouter les 3 autres d'où une couverture avec 4 sous-ensembles alors que l'on peut couvrir U avec les 3 sous-ensembles de cardinal 2.

Question 10

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

Il semble dangereux de laisser n et f en variable globale, je les donne comme paramètres.

Question 11
- On maintient une variable `vus` qui représente l'ensemble des éléments déjà couverts.
- L'objectif est que cet ensemble parvienne à l'ensemble total U représenté par $2^n - 1$ que l'on calcule avec `set2int` appliqué à la liste de n fois `True`.
- À chaque étape, on calcule, parmi les sous-ensembles de F, celui qui couvrirait le maximum d'éléments non vus et on le marque dans la liste des sous-ensembles indexée par les indices dans F. On aurait pu directement maintenir un entier en ajoutant 2^p ($p =$`ens_max`) sous la forme `couv = couv + (1 << p)`. Ici on convertit la liste en entier à la fin.

```
def glouton(n, f):
    p = len(f)
    vus = 0
    U = set2int([True]*n)
    couv = [False]*p
    while vus < U:
        ens_max = 0
        dif_max = reste(f[0], vus)
        for i in range(1, p):
            dif = reste(f[i], vus)
            if dif > dif_max:
                dif_max = dif
                ens_max = i
        couv[ens_max] = True
        vus = union(vus, f[ens_max])
    return set2int(couv)
```

Dans toute la suite :
- n désigne le cardinal de l'ensemble U,
- p désigne le nombre de parties dans F ; on a $p \leqslant 2^n$.

Pour la complexité :

- Chaque passage dans la boucle ajoute au moins un élément de plus dans l'ensemble `vus` donc il y a au plus n passages.
- Pour chaque partie dans F on effectue une opération `reste`, de complexité linéaire en n.
- Comme on effectue une `union` en plus, la complexité d'un passage de la boucle `while` est donc majorée par $np + n$.
- La complexité est ainsi un $\mathcal{O}(n^2 p)$

.

Question 12
On commence par transformer g puis on fait l'union des F_p appartenant à G et enfin on teste si cette union est égale à U.

```
def couverture(g, n, f):
    G = int2set(g)
    p = len(G)
    vus = 0
    U = set2int([True]*n)
    for i in range(p):
        if G[i]:
            vus = union(vus, f[i])
    return vus == U
```

La complexité en $\mathcal{O}(n)$ de l'union donne une complexité en $\mathcal{O}(np)$

Question 13
Il suffit de tester tous les sous ensembles de F.

```
def optimale(n, f):
    p = len(f)
    N = set2int([True]*p)
    g_opt = N
    card_opt = p
    for g in range(N):
        card = cardinal(g) # On ne le calcule qu'une fois
        if card < card_opt and couverture(g, n, f) :
            g_opt = g
            card_opt = card
    return g_opt
```

La complexité est maintenant un $\mathcal{O}(np2^p)$.
Pour F de cardinal de l'ordre de 2^n on a une complexité en $n.2^n.2^{2^n}$!