Semestre 2 – Corrigé du sujet n°2

Problème 1. Codes de Gray

1.1 Ordre lexicographique

Question 1.1.
```
def suivant(t):
    i = len(t)-1
    while i >= 0 and t[i] == 1:
        t[i] = 0
        i -= 1
    if i >= 0:
        t[i] = 1
        return True
    return False
```

Question 1.2.
```
def affiche_nuplet(t):
    for c in t:
        print(c, end='')
    print()

def affiche_tous_les_nuplets(n):
    t = [0] * n
    affiche_nuplet(t)
    while suivant(t):
        affiche_nuplet(t)
```

1.2 Ordre de Gray

Question 1.3.
```
def ajout(a, L):
    return [[a]+u for u in L]
```

Question 1.4.
```
def monte(n):
    if n == 0: return [[]]
    return ajout(0, monte(n-1)) + ajout(1, descend(n-1))

def descend(n):
    if n == 0: return [[]]
    return ajout(1, monte(n-1)) + ajout(0, descend(n-1))
```

Question 1.5. On note m_n et d_n le nombre d'appels à `monte` et `descend`. Alors $m_n = 2 + m_{n-1} + d_{n-1}$ et $d_n = 2 + m_{n-1} + d_{n-1}$. Donc on obtient aisément que $m_n = d_n$ pour tout n, et $m_n = 2 + 2m_{n-1}$. Puisque $m_0 = 0$, on a $m_1 = 2$, $m_2 = 6$, $m_3 = 14$, … On remarque que $u_n = m_n + 2$ vérifie $u_0 = 2$ et $u_n = m_n + 2 = 2m_{n-1} + 4 = 2(m_{n-1} + 2) = 2u_{n-1}$. Ainsi, $u_n = 2^{n+1}$ et $m_n = 2^{n+1} - 2$.

Question 1.6. On remarque que `monte(n)` appelle `monte(n-1)` et `descend(n-1)`. Ceux-ci appellent chacun `monte(n-2)` et `descend(n-2)` alors qu'on aurait pu ne les appeler qu'une seule fois chacun. Donc l'idée est de partir *du bas*, et de calculer au fur et à mesure les listes correspondant à `monte(k)` et `descend(k)`. On peut en déduire un algorithme itératif. Il est délicat de comparer la complexité en termes d'appels à des fonctions (puisque la nouvelle version n'est pas récursive).

```
def monte_descend(n):
    M, D = [[]], [[]]
    for k in range(1, n+1):
        M, D = ajout(0, M) + ajout(1, D), ajout(1, M) + ajout(0, D)
    return M, D
```

```
In [1]: %time _ = monte(15)
CPU times: user 219 ms, sys: 3.96 ms, total: 223 ms
Wall time: 222 ms

In [2]: %time _ = monte_descend(15)
CPU times: user 106 ms, sys: 44 µs, total: 106 ms
Wall time: 105 ms
```

1.3 Numérotation des codes de Gray

Question 1.7. Puisqu'on obtient la liste des n-uplets de Gray à partir de la liste des $(n-1)$-uplets, en ajoutant un 0 en tête, puis un 1 en tête avec la même liste à l'envers, on voit facilement que la liste des n-uplets de Gray, si on leur supprime le premier bit, est un palindrome. Ainsi, $g(2^n+r)$ et $g(2^n-1-r)$ ont la même représentation si on excepte leur premier bit. Avec $k=2^n+r$, on obtient $g(k)=2^n+g(2^n-1-r)$ puisque $g(2^n+r)$ commence par un 1 et $g(2^n-1-r)$ par un 0.

Question 1.8. Si $k=0$ ou 1, la propriété est vérifiée. Si k s'écrit $b_n\cdots b_0$ en binaire, avec $b_n=1$, et qu'on écrit $k=2^n+r$, alors r s'écrit $b_{n-1}\cdots b_0$. Donc 2^n-1-r est la différence entre $1\cdots1$ (n fois le bit 1) et r : c'est donc le nombre qui s'écrit $\bar{b}_{n-1}\dots\bar{b}_0$ où $\bar{b}_j=1$ si $b_j=0$ et inversement.

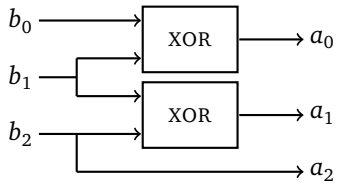
Par hypothèse de récurrence, $g(2^n-1-r)$ s'écrit $c_n\cdots c_0$ avec $c_j=\bar{b}_j\oplus\bar{b}_{j+1}$ pour tout j. Or $\bar{b}_j\oplus\bar{b}_{j+1}=b_j\oplus b_{j+1}$ (à vérifier !). Donc $c_j=a_j$ (avec la définition de l'énoncé) pour $j<n$. La question précédente permet de conclure, d'après le principe de récurrence.

Question 1.9. D'après l'indication, $g(k)=k\oplus\lfloor k/2\rfloor$ car $\lfloor k/2\rfloor$ est l'entier dont le $j^{\text{ème}}$ bit est b_{j+1}.

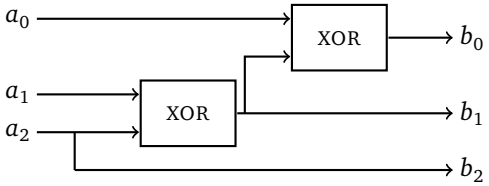
Question 1.10. On a $a_n=b_n\oplus b_{n+1}=b_n$ car $b_{n+1}=0$, donc $b_n=a_n$. Ensuite, $a_{n-1}=b_{n-1}\oplus b_n=b_{n-1}\oplus a_n$, d'où $b_{n-1}=a_{n-1}\oplus a_n$. De même, $a_{n-2}=b_{n-2}\oplus b_{n-1}$, d'où $b_{n-2}=a_{n-2}\oplus a_{n-1}\oplus a_n$. Par récurrence immédiate, $b_j=\bigoplus_{k\geq j}a_k$.

1.4 Implantation matérielle

Question 1.11. On a $a_0=b_0\oplus b_1$, $a_1=b_1\oplus b_2$ et $a_2=b_2$. Le circuit est donc celui ci-dessous.



Question 1.12. On a montré que $a_2=b_2$, $a_1=a_2\oplus a_1$ et $a_0=a_2\oplus a_1\oplus a_0$. D'où le circuit ci-dessous.



Question 1.13. On généralise facilement les circuits précédents pour obtenir des circuits à $n-1$ portes.

1.5 Parcours de sous-ensembles

Question 1.14.
```
def comb3(n):
    if n < 3: raise ValueError("il n'y a pas de 3-combinaison dans    \
            {0, ..., n-1} si n < 3")
    for c0 in range(n):
        for c1 in range(c0+1, n):
            for c2 in range(c1+1, n):
                print([c0,c1,c2])
```

Question 1.15. La première combinaison est $[0,\dots,p-1]$ et la dernière est $[n-p,\dots,n-1]$.

Question 1.16.
```
def comb_suivante(c, n):
    j = len(c)-1
    if c[j] < n-1:
        c[j] += 1
    else:
        while j > 0 and c[j-1] == c[j] - 1:
            j -= 1
        c[j-1] += 1
        while j < len(c):
            c[j] = c[j-1] + 1
            j += 1
```

Question 1.17.
```
def combinaisons(n, p):
    if n < p: raise ValueError(f"il n'y a pas de {p}-combinaison de   \
            {{0, ..., {n-1}}}")
    c = list(range(p))
    while c != list(range(n-p,n)):
        print(c)
        comb_suivante(c,n)
    print(c)
```

Question 1.18. Il suffit de représenter une combinaison par un n-uplet t tel que $t_i = 1$ si i appartient à la combinaison, et 0 sinon. Les combinaisons sont donc les n-uplets comportant exactement p bits à 1.

Question 1.19.
```
def monte_p(n, p):
    if n < p or p < 0: return []
    if n == 0: return [[]]
    return ajout(0, monte_p(n-1, p)) + ajout(1, descend_p(n-1, p-1))

def descend_p(n, p):
    if n < p or p < 0: return []
    if n == 0: return [[]]
    return ajout(1, monte_p(n-1, p-1)) + ajout(0, descend_p(n-1, p))
```

Question 1.20. On peut montrer le résultat par récurrence sur n. On remarque que le résultat n'est pas formellement correct pour $n = 1$ (donc $p \leq 1$) puisqu'un seul bit peut changer... Pour $n = 2$, le résultat est immédiat. On considère donc $n > 2$. Si les deux combinaisons consécutives commencent par le même bit, il s'agit de deux combinaisons de p ou $(p-1)$ bits parmi $n-1$, et par hypothèse de récurrence elles ne différent que par deux bits.

Sinon, par construction du code de Gray, le premier bit ne change qu'une seule fois : on a d'abord des n-uplets commençant par 0 et des n-uplets commençant par 1. Le dernier n-uplet commençant par 0 est obtenu comme la dernière combinaison de p bits parmi $(n-1)$ à laquelle on ajoute un 0 en tête, et le premier commençant par 1 est obtenu comme la dernière combinaison de $(p-1)$ bits parmi $(n-1)$ à laquelle on rajoute un 1 en tête. On doit donc déterminer le dernier n-uplet avec p bits dans le code de Gray.

La première combinaison est clairement $0\cdots01\cdots1$ avec p bits à 1. Pour la dernière, à nouveau par construction, il s'agit du premier $(n-1)$-uplet avec $p-1$ bits à 1 auquel on rajoute un 1 en tête, c'est-à-dire $10\cdots01\cdots1$ avec $(p-1)$ bits à 1 à la fin.

Ainsi, on obtient que le passage du premier bit de 0 à 1 passe du n-uplet $0\,10\cdots01\cdots1$ au n-uplet $1\,10\cdots01\cdots1$. Ainsi, les deux n-uplets diffèrent par leur premier bit, et par leur $(n-p+2)^{\text{ème}}$ qui passe de 1 à 0.

Problème 2. Implantation d'une base de données

2.1 Implantation des opérateurs de l'algèbre relationnelle en Python

Sélection avec test d'égalité à une constante

Question 2.1.
```
def SelectionConstante(table,indice,constante):
    T=[]
    for e in table:
        if e[indice]==constante:
            T.append(e)
    return T
```

Question 2.2. La boucle est effectuée n fois (n est la taille de la table). Chaque tours de boucle se fait en temps constant. la complexité est ainsi $O(n)$.

Sélection avec test d'égalité entre deux attributs

Question 2.3.
```
def SelectionEgalite(table,indice1,indice2):
    T=[]
    for e in table:
        if e[indice1]==e[indice2]:
            T.append(e)
    return T
```

Projection sur des indices

Question 2.4.
```
def ProjectionEnregistrement(enregistrement,listeIndices):
    l=[]
    for i in listeIndices:
        l.append(enregistrement[i])
    return l
```

Question 2.5.
```
def Projection(table,listeIndices):
    T=[]
    for e in table:
        T.append(ProjectionEnregistrement(e,listeIndices))
    return T
```

Produit cartésien

Question 2.6.
```
def ProduitCartesien(table1,table2):
    T=[]
    for e1 in table1:
        for e2 in table2:
            T.append(e1+e2)
    return T
```

Jointure

Question 2.7. Une première fonction prend en argument un enregistrement et un indice et renvoie l'enregistrement obtenu en supprimant l'atribut associé à l'indice.

```
def supprimer(e,i):
    l=[]
    for j in range(len(e)):
        if j!=i:
            l.append(e[j])
    return l
```

Il reste à agir comme pour le produit cartésien mais en n'ajoutant que les bons enregistrements.

```
def Jointure(table1,table2,indice1,indice2):
    T=[]
    for e1 in table1:
        for e2 in table2:
            if e1[indice1]==e2[indice2]:
                T.append(e1+supprimer(e2,indice2))
    return T
```

Question 2.8. La suppression d'une coordonnée a un coût $O(k_2)$. On itère $n_1 \times n_2$ fois et chaque étape coûte potentiellement de l'odre de k_2 opérations. la complexité est ainsi $O(n_1 n_2 k_2)$.

Distinct

Question 2.9. On utilise une fonction `egaux` testant l'égalité de deux enregistrements.

```
def egaux(e1,e2):
    for i in range(len(e1)):
        if e1[i]!=e2[i]:
            return False
    return True
```

Pour chaque enregistrement de la table, on teste s'il existe un enregistrement de numéro strictement plus grand qui lui est égal. Si ce n'est pas le cas, on l'ajoute à une table en constuction.

```
def SupprimerDoublons(table):
    T=[]
    for i in range(len(table)):
        j=i+1
        while j<len(table) and not(egaux(table[i],table[j])):
            j=j+1
        if j==len(table):
            T.append(table[i])
    return T
```

Question 2.10. Le test d'égalité d'enregistrements a un coût $O(k)$. Dans la fonction, pour chaque i, on fait au plus $n-i$ appels à la fonction d'égalité. La complexité est alors (somme des $k(n-i)$) égale à $O(kn^2)$).

2.2 Implémentation de requêtes SQL en Python

Question 2.11.
```
resultat = SelectionConstante(Trajet,1,"Rennes")
```

Question 2.12.
```
resultat = ProduitCartesien(Trajet,Vehicule)
```

Question 2.13.
```
r1 = ProduitCartesien(Trajet,Vehicule)
resultat = SelectionEgalite(r1,3,0)
```

Question 2.14. Il faut faire attention au décalage d'indice pour la seconde table (puisque l'on supprime l'attribut redondant dans la jointure)

```
r1 = Jointure(Hotel,Chambre,0,1)
resultat = Projection(r1,[1,2,4,5])
```

Question 2.15. On effectue une première jointure (table `r1`) et on ne garde, par ailleurs, que les tickets de prix demandé (table `r2`).

```
r1 = Jointure(Hotel,Trajet,2,2)
r2 = SelectionConstante(Ticket,5,"50")
```

Il faut alors associer chaque trajet de `r1`, au ticket correspondant, ce que l'on fait par une nouvelle jointure. On ne garde alors que l'attribut identifiant l'hôtel.

```
resultat = Projection(Jointure(r1,r2,3,1),[0])
```

Question 2.16. On commence par reprendre la requête précédente en supprimant les doublons.

```
r1 = Jointure(Hotel,Trajet,2,2)
r2 = SelectionConstante(Ticket,5,"50")
r3 = SupprimerDoublons(Projection(Jointure(r1,r2,3,1),[0]))
```

Par ailleurs, on ne garde de la table `Chambre` que les chambre ayant le bon prix.

```
r4 = SelectionConstante(Chambre,3,"100")
```

Il ne fautgarder de `r4` que les enregistrements dont l'attribut `IdHotel` est dans `r3`. Comme en question précédente, on peu modéliser cela avec une jointure.

```
resultat = Jointure(r4,r3,1,0)
```

2.3 Amélioration des performances

Tables triées par rapport à un indice

Question 2.17.
```
def VerifieTrie(table,indice):
    for i in range(len(table)-1):
        if table[i][indice]>table[i+1][indice]:
            return False
    return True
```

Question 2.18. On effectue une recherche dichotomique. On écrit une fonction locale `explorer` qui prend en argument deux entiers `a` et `b` compris entre 0 et $n-1$ où n est le nombre d'enregistrements. Dans l'appel `explorer(a,b)` on renvoie la table (triée) contenant les enregistrements dont l'attribut d'indice `indice` vaut enregistrement et dont le numéro est plus grand que `a` et plus petit que `b`. La fonction utilise le principe de dichotomie.

```
def SelectionConstanteTrie(table,indice,constante):
    def explorer(a,b):
        if a>b:
            return []
        else:
            c=(a+b)//2
            if table[c][indice]<constante:
                return explorer(c+1,b)
            elif table[c][indice]>constante:
                return explorer(a,c-1)
            else:
                return explorer(a,c-1)+[table[c]]+explorer(c+1,b)
    return explorer(0,len(table)-1)
```

Question 2.19. L'algorithme demandé est une (légère) modification de l'algorithme de fusion du tri-fusion. L'idée est de gérer deux indices i_1 et i_2 correspondant à une position dans chaque table, et de parcourir faire avancer i_1 ou i_2 (le plus petit des deux) tant que $i_1 \neq i_2$. Il suffit alors d'ajouter la jointure des deux éléments et d'avancer les deux indices (par hypothèse d'unicité).

```
def JointureTrie(table1,table2,indice1,indice2):
    n1,n2=len(table1),len(table2)
    i1,i2=0,0
    T=[]
    while i1<n1 and i2<n2:
        if table1[i1][indice1]>table[i2][indice2]:
            i2=i2+1
        elif table1[i1][indice1]<table[i2][indice2]:
            i1=i1+1
        else:
```

```
            T.append(table1[i1]+supprimer(table2[i2],indice2))
            i1=i1+1
            i2=i2+1
    return T
```

Question 2.20. Dans la boucle, à coup sûr l'une des variables i_1 ou i_2 est incrémentée. La boucle est donc effectuée au plus n_1+n_2 fois. Chaque itération se faisant en temps constant, la complexité est $O(n_1+n_2)$. Cette complexité est la même dans tous les cas. Elle est toujours meilleure que celle $O(n_1n_2)$ de la fonction initiale.

Utilisation d'un dictionnaire (index)

Question 2.21. On parcourt la table. Pour un enregistrement donné, on met à jour le dictionnaire soit en créant une association (si elle n'existe pas) soit en la modifiant (si elle existe).

```
def CreerDictionnaire(table,indice):
    dico={}
    for i in range(len(table)):
        e=table[i]
        if e[indice] in dico:
            dico[e[indice]].append(i)
        else:
            dico[e[indice]]=[i]
    return dico
```

Question 2.22. On regarde si la constante est une clé. Si oui, on crée la table en ne gardant que les enregistrements dont le numéro est dans la liste associée. Sinon, on renvoie une table vide.

```
def SelectionConstanteDictionnaire(table,indice,constante,dico):
    if constante in dico:
        T=[]
        for i in dico[constante]:
            T.append(table[i])
        return T
    else:
        return []
```

Question 2.23. Toutes les opérations se font en temps constant. Leur nombre dépend du nombre des itérations. Il y en a autant que d'enregistrements à sélectionner. La complexité est donc $O(m)$ où m est le nombre d'enregistrements à sélectionner. Si la taille du résultat est petite par rapport au nombre total d'enregistrements, cette fonction est plus efficace que la précédente. Si tous les enregistrements ou presque vérifient la condition, alors on ne gagne rien. Si on compte le fait qu'il faut créer initialement le dictionnaire, cette nouvelle solution pourrait même être plus coûteuse dans ce cas.

Question 2.24. Pour chaque enregistrement de `table1`, `dico2` nous indique quels enregistrements de `table2` il faut lui associer.

```
def JointureDictionnaire(table1,table2,indice1,indice2,dico2):
    T=[]
    for e in table1:
        if e[indice1] in dico2:
            for i2 in dico2[e[indice1]]:
                T.append(e+supprimer(table2[i2],indice2))
    return T
```

Question 2.25. Pour chaque enregistrement `e` de la table `table1`, on va écrire une boucle qui s'effectue au plus k_2 fois et dont chaque tour se fait en temps constant. La complexité est ainsi $O(n_1k_2)$.

Question 2.26. Si on choisit d'indexer la table numéro 1, on aura une complexité $O(k_1n_2)$. Dans la mesure où l'on crée potentiellement une table de taille k_1k_2, on pourrait choisir d'indexer la table contenant le plus d'éléments.