

Exercice 1 (6 points)

Cet exercice porte sur les tableaux, les dictionnaires, les arbres binaires, la programmation en Python et la récursivité.

Partie A

1. L'écriture binaire du code ASCII 97 du caractère 'a' est 0110.0001
2. L'appel `replique([0,0,0,1,0,1])` a pour résultat `[0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,1]`.
3. On complète les lignes demandées.

```

13     if x in nb_occ:
14         nb_occ[x] += 1
15     else:
16         nb_occ[x] = 1

```

4. On complète la fonction `majorite`.

```

10     for cle in dict.keys():
11         if dict[cle] > valeur_max:
12             cle_max = cle
13             valeur_max = dict[cle]

```

Partie B

5. On a entouré le bit qui a subi une erreur.
6. On écrit la fonction `erreur_colonne` demandée.

1	①	1
1	1	0
0	1	1

```

def erreur_colonne(mat):
    if (mat[0][0] + mat[1][0] + mat[2][0]) % 2 == 1:
        return 0
    elif (mat[0][1] + mat[1][1] + mat[2][1]) % 2 == 1:
        return 1
    return 2

```

Partie C

7. Avec un code reçu 1010000, le code de Hamming le plus proche est 1110000 qui correspond au mot 1000.
8. On complète la fonction `corriger_erreur`

```

12 def corriger_erreur(code_recu):
13     if code_recu in hamming_4_7:
14         return code_recu
15     else:
16         # Copie du code reçu avec une compréhension
17         code = [bit for bit in code_recu]
18         for indice in range(7):
19             # Inversion du bit d'indice courant
20             code[indice] = (code[indice] + 1) % 2
21             if code in hamming_4_7:
22                 return code
23         else:
24             # Réinit. du bit d'indice courant
25             code[indice] = 1 - code[indice]

```

9. L'arbre décodeur complet du code de Hamming (4,7) comporte $2^7 = 128$ feuilles.
10. On complète la fonction récursive `decode`.

```

12     if code[i] == 0:
13         return decode(arbre.gauche)
14     if code[i] == 1:
15         return decode(arbre.droit)

```

Exercice 2 (6 points)

Cet exercice porte sur la gestion des bugs, l'algorithmique, les structures de données et la programmation orientée objet.

Partie A

1. Pour un collier de 8 bonbons, ceux-ci sont mangés dans l'ordre 0, 3, 6, 2, 7, 5, 1 et le bonbon restant est le 4.
2. L'erreur `NameError` est une erreur indiquant que Python ne connaît pas l'identifiant `true`. La valeur booléenne vraie est notée `True` en Python, Bob doit faire trop de JavaScript, où les valeurs booléennes sont en minuscules.
3. On complète le code de la fonction `dernier`.

```
1 def dernier(n):
2     collier=[True]*n
3     indice=0
4     collier[indice]=False
5     for etape in range(n-1):
6         nb_bonbons_vus=0
7         while nb_bonbons_vus<3:
8             indice+=1
9             if indice==n:
10                 indice=0
11                 if collier[indice]:
12                     nb_bonbons_vus+=1
13             collier[indice]=False
14     return indice
```

Partie C

7. `pred`, `valeur` et `succ` sont des attributs de la classe `Bonbon`.
8. Après exécution du code proposée, `a` a comme valeur 1 et `b` comme valeur 2.
9. On complète le code de la fonction `creer_collier`.

```
1 def creer_collier(n):
2     premier=Bonbon(0)
3     actuel=premier
4     for i in range(1,n):
5         nouveau=Bonbon(i)
6         actuel.succ=nouveau
7         nouveau.pred=actuel
8         actuel=nouveau
9     actuel.succ=premier
10    premier.pred=actuel
11    return premier
```

10. On dessine le collier obtenu.

Le bonbon 0 a été mangé et n'appartient plus au collier.

11. Proposition C `bonbon.valeur == bonbon.succ.valeur`
12. On complète la fonction `dernier_chaine`.

```
1 def dernier_chaine(n):
2     bonbon=creer_collier(n)
3     while bonbon.valur!=bonbon.succ.valeur:
4         bonbon.pred.succ=bonbon.succ
5         bonbon.succ.pred=bonbon.pred
6         bonbon=bonbon.succ.succ.succ
7     return bonbon.valeur
```

Partie B

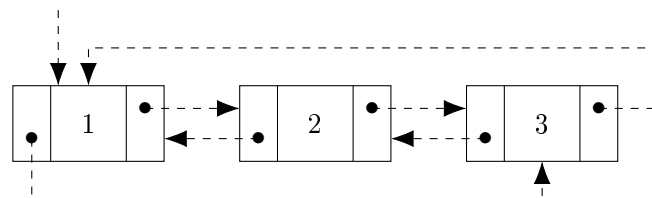
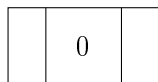
4. Une `File` est une structure LIFO.
5. L'affichage sera le suivant.

```
>>> f.affiche()
(Tête) 3 4 1 2 (Queue)
```

6. On écrit une fonction `dernier_file` qui utilise une `File`.

```
def dernier_file(n):
    f=File()
    for bonbon in range(n):
        f.enqueue(bonbon)
    while True:
        bonbon=f.dequeue()
        if f.est_vide():
            return bonbon
        f.enqueue(f.dequeue())
        f.enqueue(f.dequeue())
```

bonbon



Exercice 3 (8 points)

Cet exercice porte sur les bases de données, la programmation en Python, la récursivité et les algorithmes de parcours de graphes.

Partie A

1. La clé `id_avion` n'est pas unique dans la table `reservation` et ne peut donc pas servir de clé primaire pour cette table.
2. Le couple `(id_vol, id_passager)` peut servir de clé primaire.
3. Dans une relation, une clé étrangère permet de référencer une autre table (et donc de faire des jointures).
4. La requête `SELECT id_vol FROM vol WHERE aeroport_arr='CDG'`; renvoie les `id_vol` des vols dont l'aéroport d'arrivée est Paris-Charles de Gaulle. Avec l'extrait de table proposé, on obtient cette table :

id_vol
AI0015
AI0258
AI0292

5. La requête `SELECT ville FROM aeroport JOIN vol ON ville.id_aeroport=vol.aeroport_arr WHERE vol.aeroport_dep=CDG` ; convient.
6. On peut mettre à jour la table `passager` avec la requête `UPDATE passager SET d_total=16 WHERE id_passager=5`;
7. L'erreur commise ici est une erreur d'intégrité, la clé primaire `AI0256` est déjà utilisée pour le vol Paris-Sidney, il faut choisir une autre valeur pour la clé primaire, p.ex `INSERT INTO vol VALUES('AI1024', 'CDG', 'YUL', 6)`;

Partie B

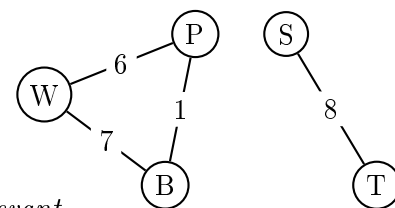
8. La valeur de `graphe_airinfo['T']['P']` est 10.
9. On écrit la fonction `vol_direct` demandée.

```
def vol_direct(graphe, ville1, ville2):  
    return ville2 in graphe[ville1]
```

10. On écrit la fonction `liste_villes_proches` demandée.

```
def liste_villes_proches(graphe, ville, d_max):  
    return [v for v in graphe[ville] if graphe[ville][v] <= d_max]
```

11. On construit le graphe représentant le réseau aérien de la compagnie *Droidevant*.
12. Proposition A : le graphe de la compagnie AirInfo est connexe.
13. La fonction `parcours` est récursive car elle s'appelle elle-même.
14. Après l'exécution du code, `visitees1` contient `['W', 'P', 'T', 'B', 'S']` et `visitees2` contient `['W', 'P', 'B']`.
15. Le parcours de graphe utilisé ici est Proposition C : un parcours en profondeur.
16. On complète la fonction `est_connexe`.



16. On complète la fonction `est_connexe`.
17. L'appel `mystere(graphe_airinfo, 'W', [], 0, 'B')` produit l'affichage ci-dessous.

```
['W', 'P', 'T', 'B'] 25  
['W', 'P', 'S', 'T', 'B'] 40
```

18. De façon générale, l'appel `mystere(graphe, ville, [], 0, arrivee)` affiche les chemins sans boucle d'une ville de départ `ville` à une ville d'arrivée `arrivee` dans le réseau représenté par le graphe `graphe`, ainsi que la distance correspondante.