# 2025 Amérique du nord jour 2

## Exercice 1

### Partie A

**1.** 01100001

**2.** [0,0,0,0,0,0,1,1,1,0,0,0,1,1,1]

**3.**

```
def nb_occurrences(tab, i):
    '''
    Renvoie un dictionnaire qui associe, à chaque élément
    apparaissant dans tab entre la position 3i
    incluse et la position 3(i + 1) exclue,
    son nombre d'occurrences.
    >>> nb_occurrences([0, 0, 1, 1, 0, 1, 0, 1, 1], 1)
    {1: 2, 0: 1}
    '''
    nb_occ = {}
    for j in range(3 * i, 3 * (i + 1)):
        x = tab[j]
        if x in nb_occ:
            nb_occ[x] = nb_occ[x] + 1
        else:
            nb_occ[x] = 1
    return nb_occ
```

**4.**

```
def majorite(dict):
    '''
    Renvoie une clé du dictionnaire dict pour laquelle la
    valeur associée est la plus grande.
    Précondition : dict est un dictionnaire dont toutes
    les valeurs sont positives.
    '''
    cle_max = None
    valeur_max = -1
    for cle in dict.keys():
        if dict[cle] > valeur_max:
            valeur_max = dict[cle]
            cle_max = cle
    return cle_max
```

## Partie B

**5.**

| | | |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 0 |
| 0 | 1 | 1 |

**6.**

```
def erreur_colonne(mat):
    for j in range(3):
        s = 0
        for i in range(3):
            s = s + mat[i][j]
        if s % 2 == 1:
            return j
```

## Partie C

**7.** `1110000` diffère de `1010000` de 1 bit. Le mot initial est donc `1000`.

**8.**

```
def corriger_erreur(code_recu):
    if code_recu in hamming_4_7:
        return code_recu
    else:
        # Copie du code reçu créée par compréhension
        code = [b for b in code_recu]
        for indice in range(7):
            # Inversion du bit d'indice courant
            code[indice] = (code[indice] + 1) % 2
            if code in hamming_4_7:
                return code
            else:
                # Réinit. du bit d'indice courant
                code[indice] = (code[indice] + 1) % 2
```

**9.** $2^7 = 128$ feuilles.

**10.**

```
def decode(arbre, code, i):
    '''
    Descend dans l'arbre binaire arbre en lisant le
    tableau code à partir de l'indice i et renvoie
    le mot étiquetant la feuille atteinte.
    Précondition : arbre est un arbre binaire
    de hauteur len(code) - i
```

```
    '''
    if i == len(code):
        return arbre.etiquette
        if code[i] == 0:
            return decode(arbre.gauche, code, i+1)
        if code[i] == 1:
            return decode(arbre.droit, code, i+1)
```

# Exercice 2

## Partie A

**1.** 0-3-6-2-7-5-1-4

**2.** `true` n'est pas définie. Il faut utiliser une majuscule : `True`.

**3.**

```
def dernier(n):
    collier = [True for i in range(8)]
    indice = 0
    collier[indice] = False
    for etape in range(n-1):
        nb_bonbons_vus = 0
        while nb_bonbons_vus < 3:
            indice += 1
            if indice >= n:
                indice = 0
            if collier[indice]:
                nb_bonbons_vus += 1
        collier[indice] = False
    return indice
```

## Partie B

**4.** FIFO

**5.** `[3, 4, 1, 2]`

**6.**

```
def dernier(n):
    f = File()
    for x in range(n):
        f.enfile(x)
    for i in range(n-1):
        f.defile()
        f.enfile(f.defile())
        f.enfile(f.defile())
    return f.defile()
```

## Partie C

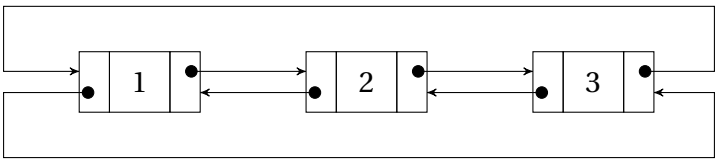
**7.** Attributs

**8.** `a = 1` et `b = 2`

**9.**

```
def creer_collier(n):
    premier = Bonbon(0)
    actuel = premier
    for i in range(1, n):
        nouveau = Bonbon(i)
        actuel.succ = nouveau
        nouveau.pred = actuel
        actuel = nouveau
    nouveau.succ = premier
    premier.pred = nouveau
    return premier
```

**10.** `bonbon` est le 1. Le 0 a été enlevé du collier.




**11.** Proposition C

**12.**

```
def dernier_chaine(n):
    bonbon = creer_collier(n)
    while bonbon.valeur != bonbon.succ.valeur:
        bonbon.pred.succ = bonbon.succ
        bonbon.succ.pred = bonbon.pred
        bonbon = bonbon.succ.succ.succ # décalage de 3 bonbons
    return bonbon.valeur
```

# Exercice 3

**1.** Il n'est pas unique. la valeur `AI0006` apparaît deux fois.

**2.** On doit utiliser une clé multiple `(id_vol, id_passager)` car un passager peut faire plusieurs réservations.

**3.** Une clé étrangère fait référence à une clé primaire d'une autre relation.

**4.**

| id_vol |
|---|
| AI0015 |
| AI0258 |
| AI0292 |

**5.**

```sql
SELECT ville
FROM aeroport
JOIN vol
ON id_aeroport = aeroport_arr
WHERE aeroport_dep = 'CDG';
```

**6.**

```sql
UPDATE passager
SET d_totale = 16
WHERE id_passager = 5;
```

**7.** `id_vol` est une clé primaire et doit donc être unique. Or la valeur `AI0256` est déjà utilisée, il faut donc une autre valeur :

```sql
INSERT INTO vol
VALUES ('AI0257', 'CDG', 'YUL', 6);
```

## Partie B

**8.** `10`

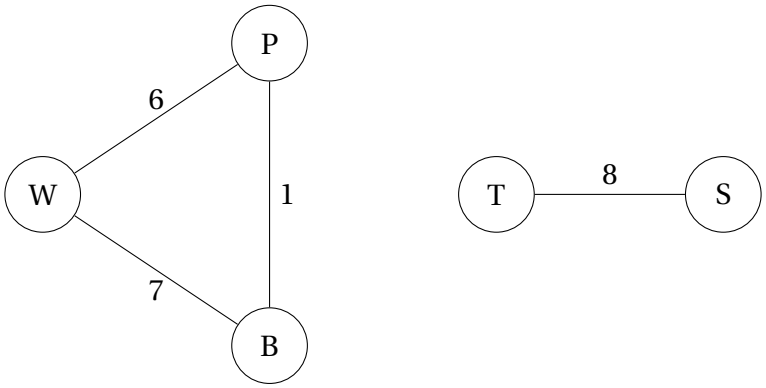
**9.**

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

**10.**

```python
def liste_villes_proches(graphe, ville, d_max):
    tab = []
    for v, d in graphe[ville].items():
        if d <= d_max:
            tab.append(v)
    return tab
```

**11.**




**12.** Proposition A

**13.** Elle s'appelle elle-même à la ligne 12.

**14.**
[Dans un dictionnaire, on ne peut pas être sur de l'ordre des clés, il y a donc plusieurs réponses possibles]
`visitees1 = ["W", "P", "T", "B", "S"]` et `visitees2 = ["W", "P", "B"]`

**15.** Proposition C

**16.**
Le graphe est connexe si on visite toutes les villes.

```
def est_connexe(graphe):
    """Vérifie si un graphe est connexe."""
    depart = ville_arbitraire(graphe)
    visitees = []
    parcours(graphe, visitees, depart)
    return len(vistees) = len(graphe)
```

**17.**

```
["W", "P", "T", "B"] 25
["W", "P", "S", "T", "B"] 40
```

**18.** Il affiche tous les chemins et leur coût entre `ville` et `arrivee`.