# 2025 Asie jour 2

## Exercice 1

### Partie A

**1.**

```
self.jour = jour
self.mois = mois
self.annee = annee
```

**2.** Le premier mai 2000.

**3.** `d = Date(19, 6, 2024)`

**4.**

```
def get_annee(self):
    return self.annee
```

**5.**

```
def set_mois(self, mois):
    self.mois = mois
```

**6.**

```
if self.est_bissextile() :
    self.nb_jours_par_mois[1] = 29
```

### Partie B

**7.**

```
def est_bissextile(self):
    quatre = self.annee % 4 == 0
    cent = self.annee % 100 == 0
    quatrecent = self.annee % 400 == 0
    return (quatre and not(cent)) or quatrecent
```

**8.** 2001 n'est pas bissextile donc $20+28+31=79$.

**9.**

```
def nb_jours_restants(self):
    j = 365
    if self.est_bissextile():
        j = 366
    return j - self.nb_jours_passes()
```

### Partie C

[Il y a une première erreur de syntaxe, il manque une majuscule :
`d_suivant = Date(1, 1, annee)`]
[Ensuite, la méthode ne renvoie pas le bon résultat pour deux dates de la même année, il y a 365 (ou 366) jours en trop dans le cas ou `other` est bien avant la date. Mais les exemples de la question suivante ne rencontrent pas ce problème.]

**10.**

— 0
— -1
— -1
— $365 + 366 = 731$

**11.**

```
def timestamp(self):
    d = Date(1, 1, 1970)
    return self.nb_jours_depuis(d) * 24 * 3600
```

## Exercice 2

**1.** Un ordonnanceur.

**2.** Il y a cinq états :
— initié
— prêt
— élu
— bloqué
— terminé

**3.** Proposition 2 : File

**4.**

```
class Processus:
    def __init__(self, PID, priorite, temps_CPU):
        self.priorite = priorite
        self.PID = PID
        self.temps_utilisation = 0
        self.temps_CPU = temps_CPU
```

**5.** Avec les premières lignes, on comprend que les files « avancent » de gauche à droite.

```
Cycle 1 : CPU=P1 liste_files=[[P3, P2], [], []]
Cycle 2 : CPU=P2 liste_files=[[P3],[P1],[]]
Cycle 3 : CPU=P3 liste_files=[[], [P2, P1],[]]
Cycle 4 : CPU=P3 liste_files=[[], [P2, P1], []]
Cycle 5 : CPU=P1 liste_files=[[], [P2], [P3]]
```

**6.** La priorité des processus présents initialement va diminuer à chaque cycle jusqu'à 4. Mais alors les processus avec un temps d'utilisation de 4 vont terminer et des nouveaux processus vont arriver avec la priorité maximale (0). Ils vont alors être exécutés et terminer sans que le processus long ne soit exécuté.
La priorité du processus long va donc rester à 4.

**7.** Le processus long ne restera plus « coincé » à la priorité 4 puisqu'il repassera à la priorité 3 et sera donc exécuté puisqu'il sera dans la file avant les nouveaux processus.

**8.**

```
def meilleur_priorite(liste_files):
    for i in range(len(liste_files)):
        if liste_files[i] != []:
            return i
    return None
```

**9.** Nous avons juste besoin de `liste.pop()` ici pour extraire le dernier élément de la liste.

```
def prioritaire(liste_files):
    mp = meilleur_priorite(liste_files)
    if mp is None:
        return None
    else:
        return liste_files[mp].pop()
```

**10.** On suppose que `p` vaut `None` s'il n'y a pas de processus en cours d'exécution. On suppose également que la fonction `gerer` renvoie le processus élu et `None` sinon.
[Il est nécessaire d'utiliser `liste.insert(i, x)` qui n'est pas courant.]

```
def gerer(p, liste_files):
    prio = prioritaire(liste_files)
    if p is None:
        return prio
    else:
        # Il n'est pas évident de savoir s'il faut le +1 ici
        if p.temps_utilisation + 1 == p.temps_CPU:
            return None
        else:
            p.temps_utilisation += 1
            mp = meilleur_priorite(liste_files)
            if mp is not None and mp <= p.priorite:
                p.priorite += 1
                if p.priorite < len(liste_files):
                    liste_files[p.priorite].insert(0, p)
                else:
                    liste_files.append([p])
                return prio
            else:
                p.priorite += 1
                return p
```

# Exercice 3

## Partie A

**1.**

```sql
SELECT nom_patient, prenom
FROM Patient
WHERE age > 60;
```

**2.**

```sql
UPDATE Symptome
SET toux='Non'
WHERE nom_patient='Heartman';
```

**3.**

```sql
SELECT COUNT(*)
FROM Diagnostic
JOIN Symptome
ON Diagnostic.nom_patient = Symptome.nom_patient
WHERE nom_maladie='Covid-19' AND toux='Oui';
```

**4.** [On ignore l'erreur `Patients` au lieu de `Patient` qui semble involontaire]
Il existe déjà un patient avec le nom `Douglas`. Or le champ `nom_patient` est une clé primaire et doit donc être unique. Cela entraîne donc une erreur.

**5.** Il faudrait utiliser `numero_secu` comme clé primaire car il est unique pour chaque patient. Il faudrait alors l'utiliser comme clé primaire et/ou étrangère dans les autres tables utilisant `nom_patient`.

## Partie B

**6.** Il est positif à la Covid-19.

**7.** L'assertion s'assure que le nœud n'est pas une feuille pour bien renvoyer un symptôme et pas un diagnostique.

**8.**

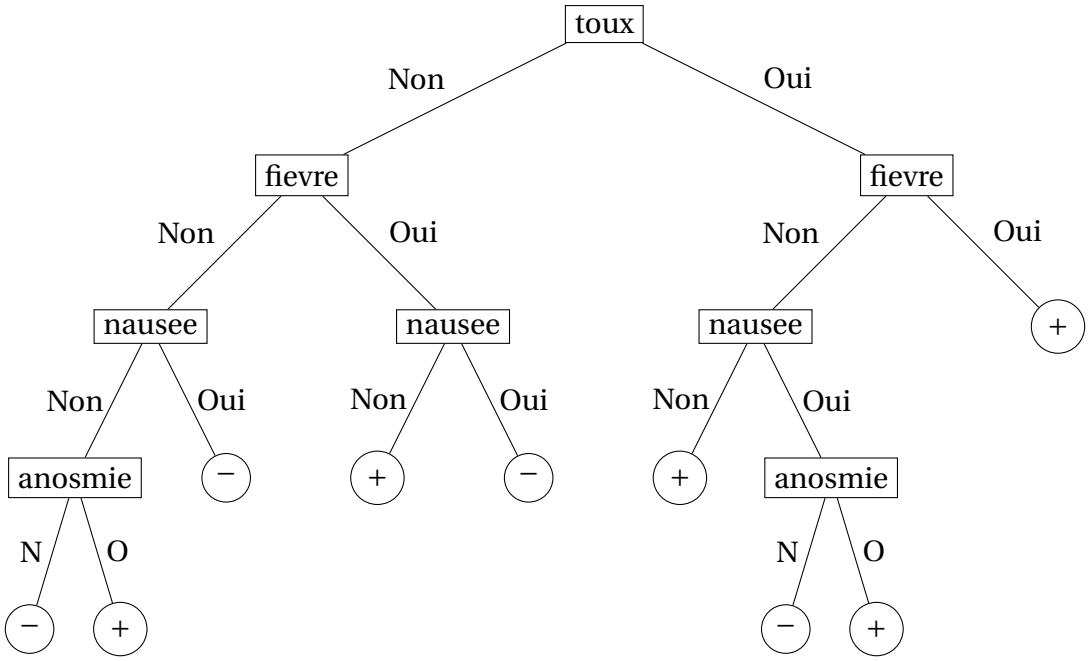
- attribut : `valeur`
- méthode : `est_feuille`

**9.**

```python
def applique(arbre, patient):
    if arbre.est_feuille():
        return arbre.valeur
    else:
        if patient[arbre.symptome()]:
            applique(arbre.droit, patient)
        else:
            applique(arbre.gauche, patient)
```

**10.** La taile est $2^5 - 1 = 31$

**11.** On suppose qu'il faut le faire récursivement :




**12.**

```
def reduire(self):
    """fonction récursive qui réduit la taille d'un arbre de
    décision sans changer les décisions prises"""
    if self.est_feuille():
        return
    self.gauche.reduire()
    self.droit.reduire()
    if self.gauche.est_feuille() and self.droit.est_feuille() \
        and self.gauche.valeur == self.droit.valeur :
        self.valeur = self.gauche.valeur
        self.gauche = None
        self.droite = None
```

**13.**

```
def verifie(num_secu):
    n = num_secu // 100
    k = num_secu % 100
    # n + k doit être multiple de 97
    return (n + k) % 97 == 0
```

**14.**

```
def cle(n):
    return 97 - (n % 97)
```