2022 Polynésie sujet 1

Exercice 1

1.a. [0.25 point] Elle est récursive car elle s'appelle elle-même.

1.b. [0.5 point] À chaque fois que la fonction `choice([True, False])` renvoie `False`, `A()` est appelée récursivement. On pourrait donc atteindre la limite de récursion de Python.

2.a. [0.5 point]

```
def A(n):
    if n <= 0 or choice([True, False]) :
        return "a"
    else:
        return "a" + A(n-1) + "a"
```

2.b. [0.5 point] Dans le pire des cas, si `choice([True, False])` renvoie `False` à chaque appel, au bout de la 50e fois on aura $n = 0$ et donc la fonction va s'arrêter.

3. [0.5 point]

- `B(0)` : `"bab"` ;
- `B(1)` : `"bbabb"` ou `"bab"` ;
- `B(2)` : `"bbbabbb"`, `"bbabb"`, `"baaab"` ou `"bab"` ;

4.a. [0.75 point]

```
def regleA(chaine):
    n = len(chaine)
    if n >= 2:
        return chaine[0] == "a" and chaine[n-1] == "a" and
               regleA(raccourcir(chaine))
    else:
        return chaine == "a"
```

4.b. [1 point]

```
def regleB(chaine):
    n = len(chaine)
    if n >= 3:
        return chaine[0] == "b" and chaine[n-1] == "b" and
               (regleA(raccourcir(chaine)) or regleB(raccourcir(chaine))
    else:
        return False
```

Exercice 2

1. [0.5 point]

Numéro du périphérique	Adresse	Opération	Réponse de l'ordonnanceur
0	10	écriture	OK
1	11	lecture	OK
2	10	lecture	ATT
3	10	écriture	ATT
0	12	lecture	OK
1	10	lecture	OK
2	10	lecture	OK
3	10	écriture	ATT

2. [0.5 point] Le périphérique 1 ne peut jamais lire à l'adresse car elle est toujours bloquée par le périphérique 0 qui demande une écriture avant.

3.a. [0.5 point]

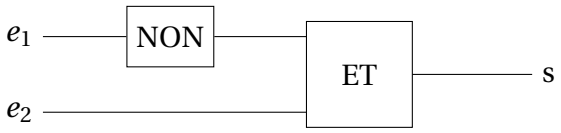
— premier tour : le périph. 0 peut écrire et le périph. 1 ne peut pas lire ;
— deuxième tour : le périph. 0 ne peut pas écrire et le périph. 1 peut lire ;
— troisième tour : le périph. 0 peut écrire et le périph. 1 ne peut pas lire ;
— quatrième tour : le périph. 0 peut écrire et le périph. 1 ne peut pas lire ;

3.b. [0.25 point] Sur les quatre tours, le périph. 0 écrit trois fois et le périph. 1 lit une fois. La proportion est donc de $\frac{1}{3}$.

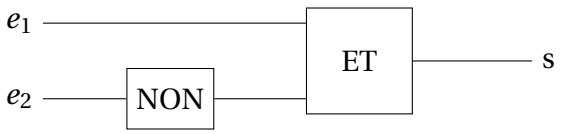
4. [0.75 point]

Tour	Numéro du périphérique	Adresse	Opération	Réponse de l'ordonnanceur	ATT_L	ATT_E
1	0	10	écriture	OK	vide	vide
1	1	10	lecture	ATT	(1,10)	vide
1	2	11	écriture	OK	(1,10)	vide
1	3	11	lecture	ATT	(1,10) (3,11)	vide
2	1	10	lecture	OK	(3,11)	vide
2	3	11	lecture	OK	vide	vide
2	0	10	écriture	ATT	vide	(0,10)
2	2	12	écriture	OK	vide	(0,10)
3	0	10	écriture	OK	vide	vide
3	1	10	lecture	ATT	(1,10)	vide
3	2	11	écriture	OK	(1,10)	vide
3	3	12	lecture	OK	(1,10)	vide

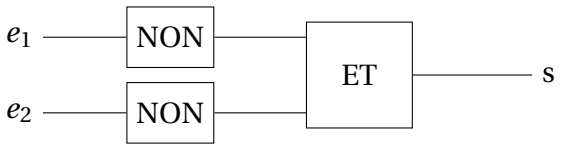
5.a. [0.5 point]



5.b. [0.5 point]



5.c. [0.5 point]



Exercice 3

1.a. [0.25 point]

```sql
SELECT ip, nompage FROM Visites;
```

1.b. [0.25 point]

```sql
SELECT DISTINCT ip FROM Visites;
```

1.c. [0.25 point]

```sql
SELECT nompage FROM Visites WHERE ip = "192.168.1.91";
```

2.a. [0.25 point] `identifiant` est la clé primaire de la table `Visites`.

2.b. [0.25 point] `identifiant` est une clé étrangère de la table `Pings`.

2.c. [0.5 point] Le SGBD va vérifier que la valeur de l'attribut `identifiant` est bien présente dans la table `Visites` avant de faire une insertion dans la table `Pings`.

3. [0.5 point]

```sql
INSERT INTO Pings
VALUES (1534, 105);
```

4.a. [0.5 point]

```sql
UPDATE Pings
SET duree = 120
WHERE identifiant = 1534;
```

4.b. [0.25 point] Les données sont transmises par internet grâce au protocole TCP/IP qui ne garantit pas l'ordre d'arrivée des paquets car le chemin emprunté peut être différent.

4.c. [0.5 point] Avec une requête d'insertion, il est possible de perdre des données si des paquets se perdent.
Avec une une requête de mise à jour, nous pouvons en plus perdre des données si elles n'arrivent pas dans l'ordre. Par exemple, si le dernier doublet (1534, 60) arrive avant le précédent (1534, 45) le temps de présence retenu (45 secondes, car il aura remplacé 60 secondes) sera alors erroné.

5. [0.5 point]

```sql
SELECT DISTINCT nompage
FROM Visites
JOIN Pings
ON Visites.identifiant = Pings.identifiant
WHERE duree > 60;
```

Exercice 4

1. [0.5 point]

```
def est_triee(self):
    if not self.est_vide() :
        e1 = self.depiler()
        while not self.est_vide():
            e2 = self.depiler()
            if e1 > e2 :
                return False
            e1 = e2
```

2.a. [0.25 point] `False` car la pile est triée en ordre inverse.

2.b. [0.25 point] La pile `A` sera représentée par `[1, 2]` les deux premiers éléments ont été dépilés.

3. [0.5 point]

```
def depileMax(self):
    assert not self.est_vide(), "Pile vide"
    q = Pile()
    maxi = self.depiler()
    while not self.est_vide() :
        elt = self.depiler()
        if maxi < elt :
            q.empiler(maxi)
            maxi = elt
        else :
            q.empiler(elt)
    while not q.est_vide():
        self.empiler(q.depiler())
    return maxi
```

4.a. [0.75 point]

- fin de la première itération :
  - B : `[9, -7, 8]`
  - q : `[4]`
  - maxi : 12
- fin de la deuxième itération :
  - B : `[9, -7]`
  - q : `[4, 8]`
  - maxi : 12
- fin de la troisième itération :
  - B : `[9]`
  - q : `[4, 8, -7]`
  - maxi : 12
- fin de la quatrième itération :
  - B : `[]`
  - q : `[4, 8, -7, 9]`
  - maxi : 12

4.b. [0.25 point]
- B : `[9, -7, 8, 4]`
- q : `[]`

4.c. [0.5 point] `[13, 4, 12]`
Si on déroule l'algorithme :
- fin de la première itération :
  - B : `[13]`
  - q : `[4]`
  - maxi : 12
- fin de la deuxième itération :
  - B : `[]`
  - q : `[4, 12]`
  - maxi : 13

Et donc, avant la ligne 14 :
- B : `[12, 4]`
- q : `[]`

L'ordre a été modifié.

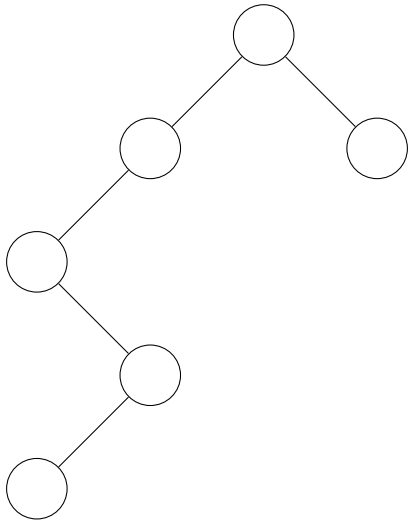
5.a. [0.75 point]
- avant la ligne 3 :
  - B : `[1, 6, 4, 3, 7, 2]`
  - q : `[]`
- avant la ligne 5 :
  - B : `[]`
  - q : `[7, 6, 4, 3, 2, 1]`
- à la fin de l'exécution de la fonction `traiter` :
  - B : `[1, 2, 3, 4, 6, 7]`
  - q : `[]`

5.a. [0.25 point] Elle trie par ordre croissant les éléments de la pile.

Exercice 5

1.a. [0.25 point] La hauteur est 2

1.b. [0.25 point]

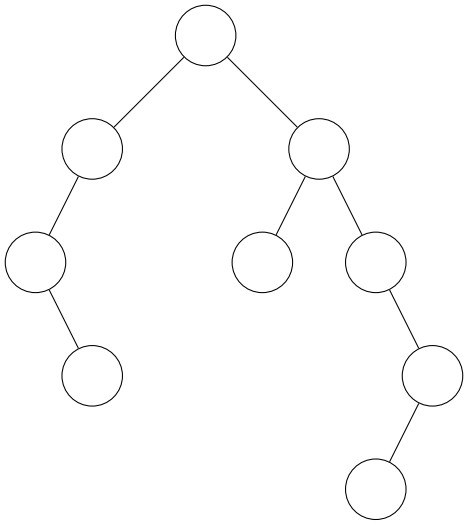


2. [0.5 point]

```
Algorithme hauteur(A):
    test d'assertion : A est supposé non vide
    si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide:
        renvoyer 0
    sinon, si sous_arbre_gauche(A) vide:
        renvoyer 1 + hauteur(sous_arbre_droit(A))
    sinon, si sous_arbre_droit(A) vide :
        renvoyer 1 + hauteur(sous_arbre_gauche(A))
    sinon:
        renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),
            hauteur(sous_arbre_droit(A)))
```

3.a. [0.25 point] Si D était vide, la hauteur de R devrait être 3. Or la hauteur de R est 4 donc D n'est pas vide.
C'est donc D qui « donne » sa hauteur à R. Ainsi la hauteur de D est 3 car $4 = 1 + 3$.

3.b. [0.25 point]

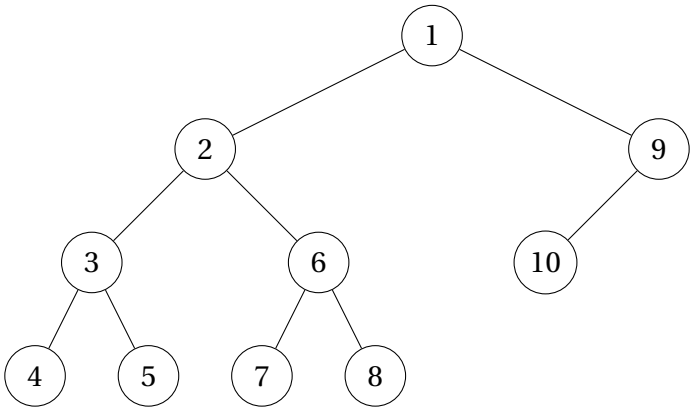


4.a. [0.25 point] On a $n = 4$ et $h = 2$. On a bien $3 \leq 4 \leq 7$.

4.b. [0.5 point] Il faut un nœud par « niveau ». On place donc la racine puis on lui ajoute un seul fils gauche ou droit. Puis pour son fils, on ajoute un seul fils etc. On continue jusqu'à la hauteur h.

4.c. [0.5 point] Il faut remplir tous les niveaux. On place la racine et on lui ajoute ses deux fils. On ajoute deux fils à chacun de ses fils etc. On continue jusqu'à la hauteur h.

5. [0.5 point]



6. [0.75 point]

```
def fabrique(h, n):
    def annexe(hauteur_max):
        if n == 0 :
            return arbre_vide()
        elif hauteur_max == 0:
            n = n - 1
            return arbre(arbre_vide(), arbre_vide())
        else:
            n = n - 1
            gauche = annexe(hauteur_max - 1)
            droite = annexe(hauteur_max - 1)
            return arbre(gauche, droite)
    return annexe(h)
```