Exercice 1 : Programmation – Récursivité

1.a. La fonction A est récursive car elle s'appelle elle-même sur la dernière ligne de la fonction.

1.b. Cette fonction récursive peut s'appeler indéfiniment si le résultat de choice est toujours False . Python fixe une profondeur de récursivité maximale de 1000 pour s'assurer que le programme s'arrête, autrement dit, Python lève une erreur si cette limite est atteinte lorsque choice renvoie False plus de 1000 fois .

2.a.

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

2.b. Si nous appelons la fonction A(n) avec n < 50, nous serons confrontés à l'un de ces cas :
- Soit elle se terminera par un choix aléatoire "Vrai" renvoyé par `choice([True, False])`.
- Soit elle se terminera par le maximum de répétitions récursives possibles avec A(n – 1) jusqu'à A(0).

3. B(0) → "bab"
B(1) → "bab", "bbabb"
B(2) → "bab", "baaab", "bbabb", "bbbabbb"

4.a.

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

4.b.

```
def regleB(n) :
    n = len(chaine)
    if (n ≥ 2) :
        return (chaine[0] == "b") and (chaine[n - 1] == "b") and
                        (regleA(chaine[1:n - 1]) or regleB(chaine[1:n - 1]))
     else :
        return False
```

Exercice 2 : Architecture Matérielle – Ordonnancement - Expressions Booléennes

1.

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. Nous savons que "*Si un périphérique demande la lecture à une adresse à laquelle on a déjà accédé en écriture, l'ordonnanceur répond "ATT" et la lecture n'a pas lieu.*". On peut donc conclure que la lecture ne se fera jamais sur l'adresse 10 puisqu'elle est occupée par le périphérique 0 en écriture.

3. a.

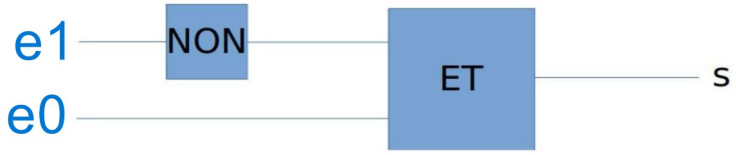
Tour	Ordre	Périphérique 0 → Adresse 10 (écriture)	Périphérique 1 → Adresse 10 (lecture)
1	0 – 1 – 2 – 3	OK	ATT
2	1 – 2 – 3 – 0	ATT	OK
3	2 – 3 – 0 – 1	OK	ATT
4	3 – 0 – 1 – 2	OK	ATT

3. b. Nous constatons que l'écriture a lieu au 1er tour et la lecture au 2ème tour donc la réponse est 1/4 soit 25%.

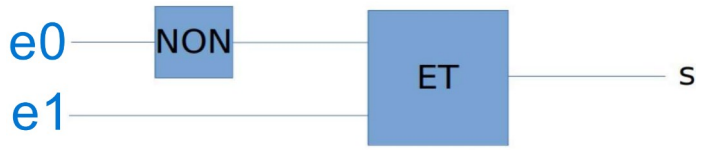
4.

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, 11) (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. Sélection du périphérique 1 :



5. b. Sélection du périphérique 2 :



5. c. Sélection du périphérique 0 :

Exercice 3 : Base de Données – SQL – Visites d'un site Web

1. a.
```sql
SELECT ip, nompage FROM Visites;
```

1. b.
```sql
SELECT DISTINCT ip FROM Visites;
```

1. c.
```sql
SELECT nompage FROM Visites WHERE ip= '192.168.1.91';
```

2. a. L'attribut "identifiant" est la clé primaire de la table "Visites".

2. b. L'attribut "identifiant" est une clé étrangère de la table "Pings".

2. c. Avant chaque insertion dans la table "Pings", le système de gestion de la base de données vérifie que la valeur de l'identifiant existe bien dans la table "Visites". C'est une des contraintes d'intégrité d'une base de données appelée la contrainte de référence. En effet, il doit être impossible de faire référence à un enregistrement qui n'existe pas.

3.
```sql
INSERT INTO Pings VALUES (1534, 105);
```
ou
```sql
INSERT INTO Pings (identifiant, duree) VALUES (1534, 105);
```

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

4. b. La structure d'internet permet que plusieurs chemins soient possibles pour faire transiter des données d'un client à un serveur. C'est ce qui assure la robustesse du système aux pannes. Cependant, ces chemins n'étant pas identiques, le temps de voyage est différent ce qui peut entraîner l'inversion de l'ordre des données entre leur départ du client et l'arrivée au serveur.

4. c. La mise à jour écrase systématiquement les données arrivées précédemment par les dernières arrivées et qui ne sont pas forcément les plus récente. Il est donc préférable d'utiliser une insertion pour conserver toutes les données reçues mais au prix d'un alourdissement de la table.

5.
```sql
SELECT Visites.nompage FROM Visites
JOIN Pings ON Visites.identifiant = Pings.identifiant
WHERE Pings.duree > 60;
```

Exercice 4 : Structures de Données – Les Piles

1.

```
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
    return True
```

2. a. L'appel `A.est_triee()` renvoie False.

2. b. Contenu de la pile A après l'exécution : `[1, 2]`

3.

```
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.

	B	q
Fin tour 1	[9, -7, 8]	[4]
Fin tour 2	[9, -7]	[4 , 8]
Fin tour 3	[9]	[4 ,8 , -7]
Fin tour 4	[]	[4, 8, -7, 9]

4. b.

	B	q
Avant la ligne 14	[9, -7, 8, 4]	[]

4. c. Par exemple si on a B = [9, 1, 2], à la fin de la première boucle «while» on a q = [1, 2], puis quand on exécute «depileMax» sur B on obtient B = [2, 1], l'ordre n'est pas préservé.

5. a.

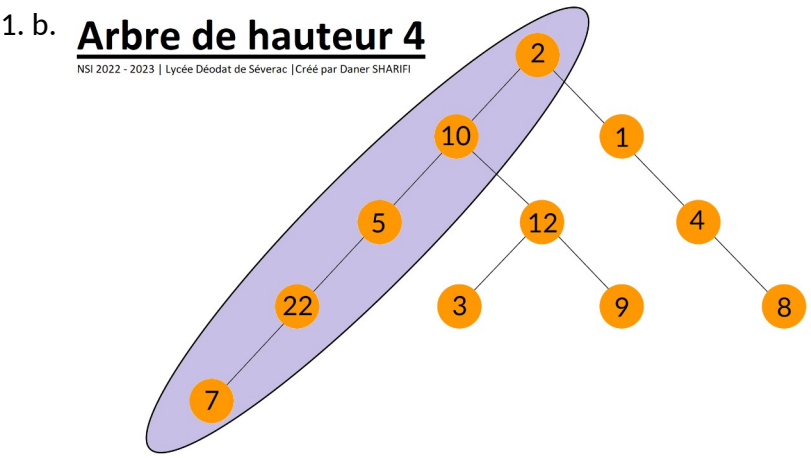
	B	q
Avant la ligne 3	[1, 6, 4, 3, 7, 2]	[]
Avant la ligne 5	[]	[7, 6, 4, 3, 2, 1]
À la fin de l'exécution de la fonction	[1, 2, 3, 4, 6, 7]	[]

5. b. Cette méthode trie les valeurs de la pile d'entrée dans l'ordre croissant.

Exercice 5 : Algorithmique – Les Arbres Binaires

1. a. D'après la définition, la hauteur de cet arbre est 2.

1. b.

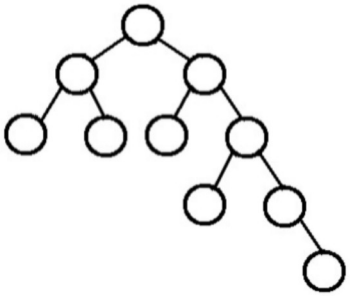


2.

```
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. On sait que : hauteur(R) = max(hauteur(G), hauteur(D)) = 4
et d'après l'énoncé : hauteur(G) = 2
soit hauteur(R) = max(2, hauteur(D)) = 4
donc il faut que hauteur(D) = 4.

3. b.



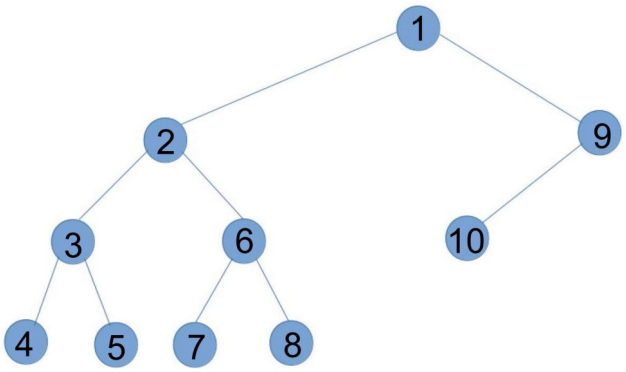
CORRECTION – NSI - 2022 Sujet Polynésie

4. a. Pour l'arbre binaire de la question 1. a. de hauteur 2 avec 4 nœuds, on obtient :
$h+1 \leqslant n \leqslant 2^{(h+1)}-1$ soit $3 \leqslant 4 \leqslant 7$
Donc l'inégalité est bien vérifiée.

4. b. Il suffit de construire un arbre binaire où chaque nœud n'a qu'un seul fils. Il s'agit d'une liste.

4. c. Dans ce cas, il faut construire un arbre binaire complet. Tous les nœuds ont deux fils et les feuilles sont toutes à la même hauteur.

5.



6.

```
def fabrique(n, h):
    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)
```