

[Index des sujets 2022](#)

## 22-NSIJ1PO1 : Corrigé

Année : **2022**

Centre : **Polynésie**

Jour : **1**

Enoncé : [!\[\]\(e3f8612927870f2e0f9f5989e6dd3064\_img.jpg\)](#)

### 1. Exercice 1

*programmation et récursivité*

1. a. Cette fonction fait appel à elle-même, elle est donc bien récursive.
- b. Si la fonction `choice` renvoie un très grand nombre de fois `False` on obtiendrait une erreur lorsque le nombre d'appel récursif maximum est dépassé.
2. a.

#### Script Python

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

- b. Le paramètre `n` décroît à chaque appel récursif de la fonction `A` or les appels récursifs s'interrompt lorsque `n` devient négatif. Donc, un appel de la forme `A(n)` avec  $n \leq 50$  s'arrête toujours.
3. • L'appel `B(0)` renvoie `"b"+A(-1)+"b"` c'est à dire `"bab"` car `A(-1)` renvoie `"a"`.
- L'appel `B(1)` peut renvoyer :
  - `"b"+A(0)+"b"` c'est à dire `"bab"`
  - `"b"+B(0)+"b"` c'est à dire `"bbabb"`.
- L'appel `B(2)` peut renvoyer :
  - `"b"+A(1)+"b"` c'est à dire `"bab"` ou `"baaab"`
  - `"b"+B(1)+"b"` c'est à dire `bbabb` ou `"bbbabbb"`

4. a.

#### Script Python

```
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"

```

b.

### Script Python

```

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

```

## 2. Exercice 2

*architecture matérielle, ordonnancement et expressions booléennes*

1. Au deuxième tour : 0 peut lire 12 (adresse non utilisée jusque là), 1 et 2 peuvent lire 10 (adresse non utilisée en écriture jusque là) et 3 ne peut pas écrire 10 puisque cette adresse a été lue auparavant.

N° 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. Le périphérique 1 ne pourra jamais lire l'adresse 10 puisque le périphérique 0 y accède en écriture avant.

3. a.

- Tour 1 : 0 peut écrire, 1 ne peut pas lire
- Tour 2 : 1 peut lire, 0 ne peut pas écrire
- Tour 3 : 0 peut écrire, 1 ne peut pas lire
- Tour 4 : 0 peut écrire, 1 ne peut pas lire

b. Durant un cycle de 4 tours, 0 écrit 3 valeurs et une seule de ces valeurs est lue par 1 la proportion est donc de  $\frac{1}{3}$ .

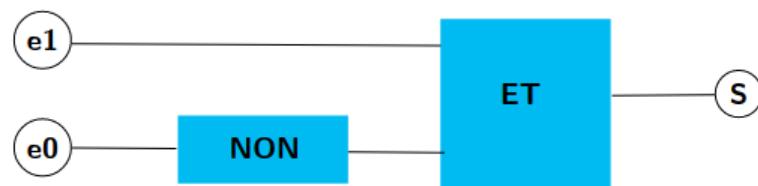
4. Tableau complété :

Tour	Pérophérique	Adresse	Opération	Réponse 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

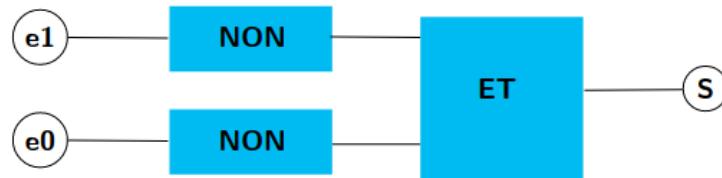
5. a. Pour le périphérique 1 : la sortie **s** est à 1 uniquement lorsque **e0** vaut 1 et **e1** vaut 0 (l'écriture binaire de 1 est 01).



b. Pour le périphérique 2 : la sortie **s** est à 1 uniquement lorsque **e0** vaut 0 et **e1** vaut 1 (l'écriture binaire de 2 est 10).



c. Pour le périphérique **0** : la sortie **s** est à 1 uniquement lorsque **e0** vaut 0 et **e1** vaut 0 (l'écriture binaire de **0** est 00).



### 3. Exercice 3

*base de données, modèle relationnel, langage SQL*

1. a.

Requête SQL

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

b.

Requête SQL

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

c.

Requête SQL

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

2. a. `identifiant` est la clé primaire de la table `Visites`

b. `identifiant` est une clé étrangère de la table `Pings`

c. Le système de gestion de base de données va vérifier :

- **la contrainte d'unicité** : deux enregistrements de la table `Visites` ne peuvent avoir le même attribut `identifiant`.
- **l'intégrité référentielle** : l'attribut `identifiant` dans `Pings` correspond à une valeur présente dans la table `Visites`

3. Requête SQL

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

4. a.

#### Requête SQL

```
UPDATE PINGS SET duree = 120 WHERE identifiant = 1534;
```

b. Dans le protocole TCP/IP les données sont scindés en paquets et chaque paquet est envoyé séparément. Les routes empruntées par les différents paquets peuvent être différents et donc l'ordre d'arrivée n'est pas forcément celui de départ. De plus, certains paquets peuvent même être perdus et donc être réexpédiés.

c. En utilisant une requête de mise à jour, on risque de mettre à jour la table avec une valeur obsolète mais arrivée plus tard qu'une valeur plus récente.

5. Requête SQL

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

## 4. Exercice 4

*structures de données, piles*

1. Script Python

```
1 def est_triee(self):
2     if not self.est_vide() :
3         e1 = self.depiler()
4         while not self.est_vide():
5             e2 = self.depiler()
6             if e1 > e2 :
7                 return False
8             e1 = e2
9     return True
```

2. a. L'appel `A.est_triee()` renvoie `False` (le sommet de la pile 4 est supérieur à l'élément situé en dessous le test ci-dessus ligne 6 est donc validé et `est_triee` renvoie `False`)

b. Après cet appel la pile `A` contient `[1,2]`, en effet seuls les deux premiers éléments ont été dépliés.

3. Script Python

```
1 def depile_max(self):
2     assert not self.est_vide(), "Pile vide"
3     q = Pile()
4     maxi = self.depiler()
5     while not self.est_vide():
6         elt = self.depiler()
7         if maxi < elt:
8             q.empiler(maxi)
9             maxi = elt
```

```

10     else :
11         q.empiler(elt)
12     while not q.est_vide():
13         self.empiler(q.depiler())
14     return maxi

```

## 4. a.

- Itération n° 1 : `B = [9, -7, 8]` et `q = [4]` (on a trouvé un nouveau maximum : 12, on empile l'ancien dans la sauvegarde)
- Itération n° 2: `B = [9, -7]` et `q = [4, 8]` (8 n'est pas un nouveau maximum, on l'empile donc dans la sauvegarde)
- Itération n° 3: `B = [9]` et `q = [4, 8, -7]`
- Itération n° 3: `B = []` et `q = [4, 8, -7, 9]`

b. La contenu de la pile de sauvegarde `q` est empilé dans la pile de départ donc à la ligne 14, `B = [9, -7, 8, 4]` et `q = []`

c. On peut prendre l'exemple de la pile `[5, 2, 3]` : \* Itération n°1 : `B = [5]` et `q = [2]` (2 n'étant pas un nouveau maximum on l'a empilé et conservé 3 comme maximum courant) \* Itération n°2 : `B = []` et `q = [2, 3]` (3 est empilé un nouveau maximum a été trouvé) \* Dans la seconde boucle while on obtient alors `B = [3, 2]` l'ordre n'est pas conservé.

5. a. Contenu des piles `B` et `q` :

- avant la ligne 3 : `B = [1, 6, 4, 3, 7, 2]` et `q = []`
- avant la ligne 5 : `B = []` et `q = [7, 6, 4, 3, 2, 1]`
- à la fin de l'exécution de `traiter()` : `B = [1, 2, 3, 4, 6, 7]` et `q = []`

b. Cette méthode permet de ranger par ordre croissant les éléments de la pile (la plus grande valeur se situe au sommet de la pile)

## 5. Exercice 5

*algorithmique, algorithme sur les arbres binaires*

## 1. a. La hauteur de cet arbre est 2.

b. Un exemple d'arbre binaire de hauteur 4 :

```

graph TD
A["A"] --> B["B"]
A --> C["C"]
B --> D["D"]
B --> E["E"]
C --> V1[" "]
C --> F["F"]
D --> G["G"]
D --> H["H"]
H --> I["I"]
H --> J["J"]
style V1 fill:#FFFFFF, stroke:#FFFFFF
linkStyle 4 stroke:#FFFFFF,stroke-width:0px

```

## 2. Texte

```

1 Algorithme hauteur(A):
2     test d'assertion : A est supposé non vide
3     si sous_arbre_gauche(A) vide et sous_arbre_droit(A) vide:
4         renvoyer 0
5     sinon, si sous_arbre_gauche(A) vide:
6         renvoyer 1 + hauteur(sous_arbre_droit(A))
7     sinon, si sous_arbre_droit(A) vide :
8         renvoyer renvoyer 1 + hauteur(sous_arbre_gauche(A))
9     sinon:
10        renvoyer 1 + max(hauteur(sous_arbre_gauche(A)),hauteur(sous_arbre_droit(A)))

```

3. a. Si D était vide, la hauteur de R serait de 3 (1 + celle de son sous arbre gauche). Donc D n'est pas vide et sa hauteur est 3.

b. Illustration par un exemple :

```

graph TD
R["R"] --> G1["G1"]
R["R"] --> D1["D1"]
subgraph Sous arbre droit de hauteur 3
D1 --> D2["D2"]
D1 --> D3["D3"]
D2 --> D4["D4"]
D2 --> V1[" "]
D4 --> D5["D5"]
D4 --> V2[" "]
end
subgraph Sous arbre gauche de hauteur 2
G1["G1"] --> G2["G2"]
G1 --> G3["G3"]
G2 --> G4["G4"]
G2 --> G5["G5"]
end
style V1 fill:#FFFFFF, stroke:#FFFFFF
style V2 fill:#FFFFFF, stroke:#FFFFFF
linkStyle 5 stroke:#FFFFFF,stroke-width:0px
linkStyle 7 stroke:#FFFFFF,stroke-width:0px

```

4. a. Sur l'arbre binaire de la question 1.a,  $h = 2$  et  $n = 4$ , et on a bien  $3 \leq 4 \leq 2^3 - 1$ . Les inégalités sont donc vérifiées

b. Il suffit que chaque noeud (sauf la feuille) ait un seul et unique fils . Par exemple pour  $h = 3$  :

```

graph TD
S1["S1"] --> S2["S2"]
S1 --> V1[" "]
S2["S2"] --> S3["S3"]
S2 --> V2[" "]
S3["S3"] --> S4["S4"]
S3 --> V3[" "]
style V1 fill:#FFFFFF, stroke:#FFFFFF
style V2 fill:#FFFFFF, stroke:#FFFFFF
style V3 fill:#FFFFFF, stroke:#FFFFFF
linkStyle 1 stroke:#FFFFFF,stroke-width:0px
linkStyle 3 stroke:#FFFFFF,stroke-width:0px
linkStyle 5 stroke:#FFFFFF,stroke-width:0px

```

c. Il suffit que chaque noeud (sauf les feuilles) ait deux fils. Par exemple pour  $h = 2$  :

```
graph TD
S1["S1"] --> S2["S2"]
S1 --> S3["S3"]
S2 --> S4["S4"]
S2 --> S5["S5"]
S3 --> S6["S6"]
S3 --> S7["S7"]
```

5.

```
graph TD
S1["S1"] --> S2["S2"]
S2 --> S3["S3"]
S3 --> S4["S4"]
S3 --> S5["S5"]
S2 --> S6["S6"]
S6 --> S7["S7"]
S6 --> S8["S8"]
S1 --> S9["S9"]
S9 --> S10["S10"]
S9 --> V1[" "]
style V1 fill:#FFFFFF, stroke:#FFFFFF
linkStyle 9 stroke:#FFFFFF,stroke-width:0px
```

6.

### Script Python

```
1 def fabrique(h, n):
2     def annexe(hauteur_max):
3         if n == 0 :
4             return arbre_vide()
5         elif hauteur_max == 0:
6             n = n - 1
7             return arbre(arbre(vide),arbre(vide))
8         else:
9             n = n - 1
10            gauche = annexe(hauteur_max - 1)
11            droite = annexe(hauteur_max - 1)
12            return arbre(gauche, droite)
13    return annexe(h)
```

### Bug

Le code donné ne fonctionne pas en l'état, en effet la variable `n` n'est pas accessible depuis le corps de la fonction `annexe`. Pour cela, il faut ajouter `non local n`