Année scolaire 2023-2024

Classe de Terminale
Spécialité NSI
Exercices avec leurs corrigés

Didier Pihoué

Structures de données

EXERCICE 1 ☞

En géométrie, un point du plan peut être représenté par deux coordonnées, son abscisse et son ordonnée, dès lors qu'a été introduit un repère du Plan.

Dans cet exercice, vous allez programmer quelques opérations sur des points du plan ainsi représentés une fois choisi un type Python pour modéliser les deux coordonnées.

Il s'agira à chaque fois de :

- créer un point
- calculer la distance d'un point à l'origine du repère
- calculer la distance entre deux points
- calculer l'image d'un point par une translation, le vecteur étant modélisé comme le point.

1. Dans cette question, on choisit le type `tuple` avec deux éléments.
   a. Complétez les codes des fonctions du fichier `nsi_points_1.py`.
   b. Testez vos codes.
2. Dans cette question, vous choisissez un autre type Python pour modéliser un point.
   a. Complétez les signatures et les codes des fonctions du fichier `nsi_points_2.py`.
   b. Testez vos codes.
3. Comparez les deux implémentations.

EXERCICE 2 ☞

En programmation ou en sciences on écrit des expressions **parenthésées**. Le terme **parenthèse** est compris ici au sens large, il peut s'agir ainsi de l'un des six caractères suivants

`'('` ; `')'` ; `'['` ; `']'` ; `'{'` ; `'}'`

L'analyse de la syntaxe d'une expression demande en particulier d'en vérifier le bon parenthésage. Autrement dit de contrôler que toute parenthèse ouvrante est associée à sa parenthèse fermante et réciproquement.

L'objectif de cet exercice est de coder une fonction qui vérifie le bon parenthésage d'une expression fournie comme chaîne de caractères et qui renvoie `True` si c'est le cas ou `False` sinon.

1. Donnez quelques exemples variés d'expressions correctement parenthésées en langage Python.
2. Identifiez les différents cas d'erreur.
3. Un algorithme d'analyse parcourt l'expression caractère par caractère, mémorise les parenthèses ouvrantes rencontrées et vérifie que chaque parenthèse fermante rencontrée est bien associée à la dernière parenthèse ouvrante mémorisée pour la fermer. Il n'est alors plus nécessaire de garder celle-ci en mémoire. À la fin du parcours, il ne doit donc plus y avoir de parenthèse ouvrante en mémoire.

Une structure de données bien adaptée pour gérer cette mémoire des parenthèses ouvrantes est une Pile, comme une pile d'assiettes. Les assiettes sont empilées les unes sur les autres et on ne s'autorise qu'à pouvoir retirer l'assiette située au sommet de la pile, on dit « dépiler ».

a. Expliquez pourquoi un tableau dynamique est bien adapté pour implémenter une structure de données Pile en Python.

b. Complétez les codes des fonctions du fichier `nsi_pile_1.py` afin de réaliser une telle implémentation.

c. Proposez une série d'instructions bien choisies pour permettre de tester vos codes.

d. Exprimez en langage naturel l'algorithme de parcours et de contrôle des parenthèses qui exploite une structure de données Pile.

Contrôlez mentalement sa cohérence et sa faisabilité.

e. Pour implémenter l'algorithme, complétez le code de la fonction `analyse_expression` qui se trouve dans le fichier `nsi_expressionParenthesee.py`.

f. Pour tester votre code, suivez la démarche suivante :

① Récupérez le fichier `nsi_testsExpressionParenthesee.py` et enregistrez-le dans le même répertoire que `nsi_expressionParenthesee.py`.

② Dans le menu de Thonny choisissez Outils puis Gérer les paquets.

③ Recherchez le paquet pytest et installez-le.

④ Fermez le gestionnaire de paquets.

⑤ Dans le menu de Thonny choisissez Outils puis Ouvrir la console du système.

⑥ Dans cette console, saisissez `pytest nsi_testsExpressionParenthesee.py`

⑦ Corrigez votre code et relancez l'instruction précédente tant que des erreurs sont signalées.

EXERCICE 3 ☞

La structure de Pile a été introduite dans l'exercice précédent mais avec une taille non limitée, cependant et concrètement la taille d'une Pile est contrainte par un maximum autorisé.

L'objectif de cet exercice est de réaliser l'implémentation d'une structure de Pile avec une taille fixée lors de sa création.

1. Listez les effets induits par cette contrainte et leurs conséquences en terme d'implémentation.

2. Complétez les codes des fonctions du fichier `nsi_pile_2.py` afin de réaliser l'implémentation demandée.

3. Proposez une série d'instructions qui permet de tester toutes les fonctions du fichier.

4. Rédigez l'interface de cette structure de données sous la forme d'un tableau dont la ligne de titre et la première ligne sont celles-ci :

Opération	Description
`creer_pile(n)`	Cette fonction renvoie une pile vide dont la taille maximale est égale à la valeur de `n`. Exemple : `p = creer_pile(10)`

EXERCICE 4 ☞

Une structure de File se comporte comme une file d'attente : un nouvel élément est placé en queue et le premier élément à sortir de la File est le premier à y être entré.

L'objectif de cet exercice est d'implémenter une structure de données de File avec un tableau informatique de taille maximale fixée comme dans l'exercice précédent pour la Pile.

On donne l'interface de cette structure de données.

Opération	Description
`creer_file(n)`	Cette fonction renvoie une file vide dont la taille maximale est égale à la valeur de `n`. Exemple : `maFile = creer_file(10)`
`est_file_vide(maFile)`	Cette fonction renvoie `True` si `maFile` est vide ou `False` sinon.
`est_file_pleine(maFile)`	Cette fonction renvoie `True` si `maFile` est pleine ou `False` sinon.
`taille_file(maFile)`	Cette fonction renvoie le nombre d'éléments de `maFile`.
`enfiler(elt, maFile)`	Cette fonction renvoie une nouvelle file après avoir enfiler `elt` dans `maFile` si celle-ci n'est pas pleine. Affiche une erreur d'assertion sinon. Exemple : `maFile = creer_file(3)` `maFile = enfiler("truc", maFile)`

`defiler(maFile)`	Cette fonction supprime le premier élément de la file si celle-ci n'est pas vide et renvoie une nouvelle file. Affiche une erreur d'assertion sinon.
	Exemple : `maFile = creer_file(3)`
	`maFile = enfiler("truc", maFile)`
	`maFile = enfiler("Bob", maFile)`
	`maFile = defiler(maFile)`

`tete(maFile)`	Cette fonction renvoie la valeur du premier élément de la file `maFile` si elle n'est pas vide. Affiche une erreur d'assertion sinon.
`afficher_file(maFile)`	Cette fonction renvoie une chaîne de caractères représentant les éléments de la file `maFile` séparés par un caractère `'|'`.

1. Complétez les codes des fonctions du fichier `nsi_file_1.py` afin de réaliser l'implémentation demandée.
2. Proposez une série d'instructions qui permet de tester toutes les fonctions du fichier.
3. Comparez cette implémentation à celle de la Pile de l'exercice précédent.

EXERCICE 5

Une autre implémentation pour la structure de file repose sur l'utilisation de deux piles :

- une pile pour l'opération d'ajout d'un élément,
- une autre pile pour l'opération de retrait du premier élément enfilé.

Cette méthode s'appuie sur la discipline suivante pour les retraits :

- si la pile de retrait est vide alors on dépile toute la pile d'ajout dans la pile de retrait puis on extrait le sommet de la pile de retrait,
- si la pile de retrait n'est pas vide alors on dépile son sommet.

On considère l'implémentation de Pile réalisée avec un tableau dynamique en reprenant les fonctions du fichier `nsi_pile_1.py`.

1. Sur papier, représentez les états successifs des deux piles qui modélisent une file pour l'enchaînement des instructions suivantes.

```
>>>maFile = creer_file()
>>>enfiler(5, maFile);enfiler(3, maFile)
>>>defiler(maFile)
>>>enfiler(7, maFile);enfiler(9, maFile)
>>>defiler(maFile)
>>>enfiler(1, maFile)
>>>defiler(maFile)
```

2. Complétez les codes des fonctions du fichier `nsi_file_2.py` afin de réaliser l'implémentation demandée.
3. Proposez une série d'instructions qui permet de tester toutes les fonctions du fichier.
4. Comparez cette implémentation à celle de l'exercice précédent.

CORRIGÉ EXERCICE 1

Une structure de données adaptée

Cet exercice illustre la nécessité d'introduire parfois une structure de données adaptée à la situation. Cette structure est ici construite à partir d'un type construit de Python. Quelques opérations sont envisagées pour manipuler la structure de données introduite.

1. Le type construit de Python est imposé. Cela permet de revoir quelques opérations sur les objets `tuple`.
   a. Voir le fichier avec les corrigés.
   b. On choisit une série d'instructions permettant de contrôler chaque fonction.

```
>>> pt1 = creer_point(3, 8)  # point de coordonnées (3, 8)
>>> pt2 = creer_point(-1.2, 3.74)  # point de coordonnées (-...
    ...1.2, 3.74)
>>> vec = creer_point(4, -9)  # vecteur de coordonnées (4, -...
    ...9)
>>> distance_origine(pt1)  # distance de O à pt1
8.54400374531753
>>> distance(pt1, pt2)  # distance entre les points pt1 et ...
    ...pt2
5.982273815197696
>>> pt3 = image_translation(pt1, vec)  # image de pt1 par la ...
    ...translation de vecteur vec
>>> pt3
(7, -1)
>>>
```

2. Ici on choisit un objet de type construit `list`, c'est-à-dire un tableau dynamique Python, mais exploité ici comme un tableau simple de taille 2.
   a. Voir le fichier avec les corrigés.
   b. On utilise la même série d'instructions pour contrôler.
3. On observe que l'utilisation des opérations ne dépend pas de leur implémentation. Ainsi, comme les opérations portent le même nom dans chacune des deux implémentations, l'exploitation est la même.
   Il n'y a pas d'écart notable en temps d'exécution car les deux implémentation comportent les mêmes types d'instructions. Il en va de même pour l'occupation de la mémoire.

CORRIGÉ EXERCICE 2

Résolution d'un problème nécessitant l'introduction d'une structure de données

1. On peut écrire des opérations arithmétiques ou des affectations portant sur des valeurs de type `tuple` ou `list` ou `dict`. Un exemple complet peut être

```
dico = 'a' : [2, 5], 'b' : (-5, 7), 'c' : [(1, 4), 6]
```

2. On identifie deux types d'erreur possible, une parenthèse ouvrante sans sa fermante ou une parenthèse fermante sans sa parenthèse ouvrante.
3. On commence par réaliser l'algorithme sur chacun des exemples, y compris en introduisant des erreurs, afin de se l'approprier en pensée.
   La structure de données Pile.

a. La taille d'un tableau dynamique n'est pas fixée a priori, ce qui semble aussi le cas pour une Pile.

La méthode `append()` permet d'ajouter un élément en dernier dans un tableau dynamique tandis que `pop()` renvoie le dernier élément en l'enlevant.

Enfin, comme la fonction `len` renvoie le nombre d'éléments d'un tableau dynamique, l'élément d'indice `len(atb) - 1` est le dernier élément du tableau `tab`.

Ainsi, un tableau dynamique Python est bien adapté pour implémenté une Pile.

b. Voir les codes du fichier corrigé.

c. Une série d'instructions pour créer une Pile vide, empiler, dépiler et provoquer une erreur d'assertion.

```
>>> maPile = creer_pile()  # une Pile vide nommée maPile
>>> est_pile_vide(maPile)  # pour tester la fonction
True
>>> empiler(9, maPile)
>>> empiler(-5, maPile)
>>> empiler(3.4, maPile)  # 3 éléments empilés, le sommet est 3.4
>>> taille_pile(maPile)  # on attend 3
3
>>> val = sommet(maPile)  # consultation avec affectation
>>> val
3.4
>>> taille_pile(maPile)  # on attend encore 3
3
>>> depiler(maPile)  # on attend 3.4
3.4
>>> depiler(maPile)  # on attend -5
-5
>>> depiler(maPile)  # on attend 9
9
>>> depiler(maPile)  # on attend une erreur d'assertion
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/home/didier/Documents/Lycee/Exercices/Codes/...
   ...nsi_pile_1_corrige.py", line 27, in depiler
    assert not est_pile_vide(pile), "La pile est vide !"
AssertionError: La pile est vide !
>>>
```

d. Voici une expression possible de l'algorithme en langage naturel.

```
Créer une Pile vide
Pour chaque caractère c de l'expression Faire
  Si c est une parenthèse ouvrante Alors
    On empile c
  Sinon Si c est une parenthèse fermante Alors
    Si la Pile est vide ou le sommet n'est pas la bonne ..
   ..ouvrante Alors
      Renvoyer Faux
    Sinon
      Dépiler
    Fin Si
  Fin Si
Fin Pour
Si la Pile est vide Alors
  Renvoyer Vrai
Sinon
  Renvoyer Faux
```

En s'y prenant en plusieurs fois et en le testant avec les exemples, on se l'approprie complètement ainsi que la structure de données Pile.

e. Il ne reste plus qu'à se concentrer sur la syntaxe du langage Python.
Voir le fichier avec le code corrigé.

f. Pour tester en grand et se préparer aux projets.

CORRIGÉ EXERCICE 3

Une autre implémentation pour une Pile

Il ne faut pas oublier qu'un tableau informatique est une structure immuable. Certaines opérations sur la structure de données Pile vont donc renvoyer un nouveau tableau. Le choix d'un `tuple` pour définir la structure impose le respect de cette contrainte.

1. Choisir un tableau de taille fixée et non un tableau dynamique nécessite de maintenir un indice d'insertion en plus du tableau pour savoir si la Pile est pleine ou à quel indice empiler. L'immuabilité des tableaux informatiques conduit aussi à systématiquement renvoyer une nouvelle Pile plutôt qu'à modifier en place.
2. Voir le fichier avec le code corrigé.
3. On va tester aussi le cas où la Pile est pleine.

```
>>> maPile = creer_pile(2)  # une Pile vide nommée maPile de taille ...
   ...maximum 2
>>> est_pile_vide(maPile)  # pour tester la fonction
True
>>> maPile = empiler(9, maPile)  # nouvelle valeur pour maPile
>>> maPile = empiler(5, maPile)  # nouvelle valeur pour maPile
>>> maPile = empiler(-3.4, maPile)  # on attend une erreur d'...
   ...assertion
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/home/didier/Documents/Lycee/Exercices/Codes/...
   ...nsi_pile_2_corrige.py", line 34, in empiler
    assert not est_pile_pleine(pile), "La pile est pleine !"
AssertionError: La pile est pleine !
```

```
>>> sommet(maPile)  # consultation du sommet de la Pile, on attend 5
5
>>> maPile = depiler(maPile)  # nouvelle valeur pour maPile
>>> taille_pile(maPile)  # on attend 1
1
>>> maPile = depiler(maPile)  # nouvelle valeur pour maPile
>>> maPile = depiler(maPile)  # on attend une erreur d'assertion
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/home/didier/Documents/Lycee/Exercices/Codes/...
   ...nsi_pile_2_corrige.py", line 44, in depiler
    assert not est_pile_vide(pile), "La pile est vide !"
AssertionError: La pile est vide !
>>>
```

4. L'interface est destinée à présenter comment utiliser la structure de données sans présenter d'élément de l'implémentation.

Opération	Description
`creer_pile(n)`	Cette fonction renvoie une pile vide dont la taille maximale est égale à la valeur de `n`. Exemple : `p = creer_pile(10)`
`est_pile_vide(pile)`	Cette fonction renvoie `True` si pile est vide ou `False` sinon.
`est_pile_pleine(pile)`	Cette fonction renvoie `True` si pile est pleine ou `False` sinon.
`taille_pile(pile)`	Cette fonction renvoie le nombre d'éléments de pile.
`empiler(elt, pile)`	Cette fonction renvoie une nouvelle pile après avoir empiler `elt` dans `pile` si celle-ci n'est pas pleine. Affiche une erreur d'assertion sinon. Exemple : `maPile = creer_pile(3)` `maPile = empiler("truc", maPile)`

`depiler(pile)`	Cette fonction supprime le sommet de la pile si celle-ci n'est pas vide et renvoie une nouvelle pile. Affiche une erreur d'assertion sinon.
	Exemple : `maPile = creer_pile(3)`
	`maPile = empiler("truc", maPile)`
	`maPile = empiler("Bob", maPile)`
	`maPile = depiler(maPile)`

`sommet(pile)`	Cette fonction renvoie la valeur du sommet de la pile si elle n'est pas vide. Affiche une erreur d'assertion sinon.
`sommet(pile)`	Cette fonction renvoie la valeur du sommet de la pile si elle n'est pas vide. Affiche une erreur d'assertion sinon.
`afficher_pile(pile)`	Cette fonction renvoie une chaîne de caractères représentant les éléments de la pile séparés par un caractère `'|'`.

CORRIGÉ EXERCICE 4

Une première implémentation d'une File

1. Il ne faut pas hésiter à effectuer des allers et retours avec les codes des deux exercices portant sur la structure de Pile afin de bien mobiliser les opérations sur les variables de type `tuple` ou `list`.

Voir les codes dans le fichier corrigé.

2. Comme dans l'exercice précédent, on teste toutes les configurations possibles.

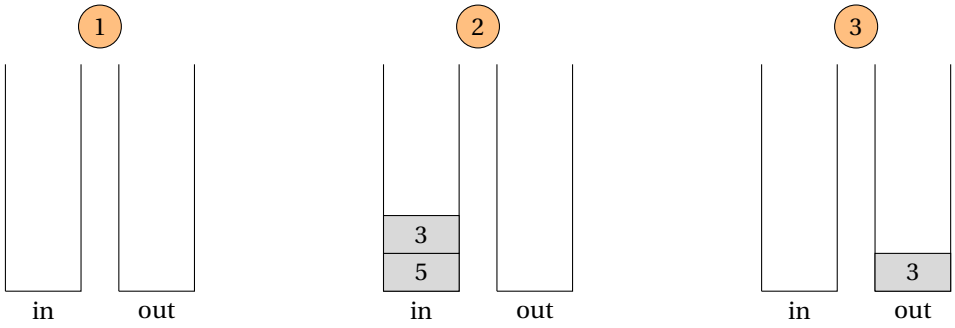
```
>>> maFile = creer_file(3)  # file vide de taille maximale égale à 3
>>> est_file_vide(maFile)  # on attend True
True
>>> est_file_pleine(maFile)  # on attend False
False
>>> defiler(maFile)  # une erreur d'assertion
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/home/didier/Documents/Lycee/Exercices/Codes/...
   ...nsi_file_1_corrige.py", line 45, in defiler
    assert not est_file_vide(file), "La file est vide !"
AssertionError: La file est vide !
>>> taille_file(maFile)  # on attend 0
0
>>> maFile = enfiler("Ada", maFile)
>>> maFile = enfiler("Alan", maFile)
>>> maFile = enfiler("Grace", maFile)
>>> est_file_pleine(maFile)  # on attend True
True
>>> tete(maFile)  # on attend "Ada" (Lovelace)
'Ada'
>>> maFile = defiler(maFile)
>>> taille_file(maFile)  # on attend 2
2
>>> afficher_file(maFile)  # pour voir
'| Grace | Alan |'
>>> maFile = enfiler('Alonzo', maFile)
>>> taille_file(maFile)  # on attend 3
3
>>> afficher_file(maFile)  # pour voir
'| Alonzo | Grace | Alan |'
>>> tete(maFile)  # on attend "Alan" (Turing)
'Alan'
>>> maFile = enfiler('John', maFile)  # on attend une erreur d'...
   ...assertion
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
  File "/home/didier/Documents/Lycee/Exercices/Codes/...
   ...nsi_file_1_corrige.py", line 35, in enfiler
    assert not est_file_pleine(file), "La file est pleine !"
AssertionError: La file est pleine !
>>>
```

3. On note une forte proximité entre les deux interfaces, cependant, alors que toutes les opérations sur une Pile se réalisent en temps constant, l'opération `defiler()` de la structure de File se réalise dans un temps proportionnel au nombre d'éléments présents dans la file.

CORRIGÉ EXERCICE 5

Une autre implémentation pour une File

1. Représentons l'état des deux piles pour les trois premières instructions.

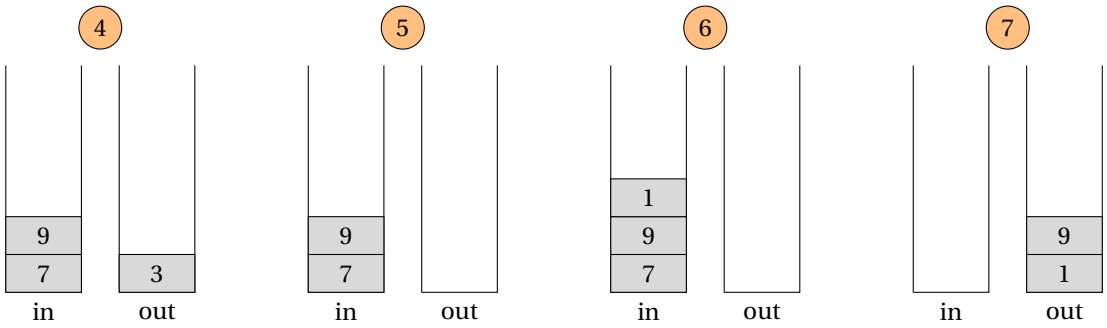


① Création des deux piles vides qui représentent la file.

② On empile 5 puis 3 dans la pile des ajouts.

③ La pile des retraits est vide donc on dépile 3 puis 5 de la pile des ajouts et on les empile dans la pile des retraits. Enfin on dépile la pile des retraits en retirant 5 qui était bien la tête de la file.

Représentons maintenant les états des deux piles pour les quatre instructions suivantes.



④ On empile 7 puis 9 dans la pile des ajouts.

⑤ On dépile 3 de la pile des retraits.

⑥ On empile 1 dans la pile des ajouts.

⑦ La pile des retraits est vide donc on dépile 1 puis 9 puis 7 de la pile des ajouts et on les empile dans la pile des retraits. Enfin on dépile la pile des retraits en retirant 7 qui était bien la tête de la file.

Ces représentations aident à comprendre comment jouent les deux piles avec la discipline de retrait.

2. On importe toutes les fonctions du fichier de structure de Pile par tableau dynamique et on utilise les opérations sur cette structure pour implémenter une File.

Une difficulté tient au caractère mutable d'un tableau dynamique Python, car il faut garder en tête que les deux piles vont être modifiées en place.

Pour « transpiler », on répète l'opération « dépiler et empiler dans la pile de retrait » autant de fois qu'il y a d'éléments dans la pile d'ajout. Une boucle `Tant que` est bien adaptée mais on peut aussi mobiliser une boucle `Pour` sur la taille de la pile d'ajout.

La file est vide si et seulement si les deux piles le sont.

La taille de la file est la somme des tailles des deux piles.

Pour enfiler, il suffit d'empiler dans la pile d'ajout.

Pour défiler, on provoque une erreur d'assertion si la file est vide puis on dépile la pile de retrait en testant avant si elle est vide et dans ce cas on mobilise la fonction utilitaire pour transpiler.

Le code de la fonction `tete()` qui est fourni donne les indications pour implémenter `defiler()`.

Voir les codes dans le fichier corrigé.

3. On procède comme dans l'exercice précédent mais en intégrant le caractère mutable de la structure de données.

```
>>> maFile = creer_file()  # une file vide nommée maFile
>>> est_file_vide(maFile)  # on attend True
True
>>> taille_file(maFile)  # on attend 0
0
>>> enfiler(5, maFile)
>>> enfiler(3, maFile)
>>> taille_file(maFile)  # on attend 2
2
>>> tete(maFile)  # on attend 5
5
>>> enfiler(7, maFile)
>>> enfiler(9, maFile)
>>> afficher_file(maFile)  # pour voir l'ordre
'| 9 | 7 | 3 | 5 |'
>>> taille_file(maFile)  # on attend 4
4
>>> defiler(maFile)
5
>>> defiler(maFile)
3
>>> defiler(maFile)  # pour contôler transpiler
7
>>> taille_file(maFile)  # on attend 1
1
>>>
```

4. Dans le pire des cas, autrement dit si on a enfilé tous les éléments pour tous les défiler ensuite, la fonction `defiler` a une complexité en **temps linéaire** de la taille de la file comme pour l'exercice précédent. Cependant, ce n'est pas la pratique la plus courante. En moyenne, la performance sera donc meilleure car la pile de retrait ne sera pas toujours vide, et alors défiler s'effectue en **temps constant**, et quand elle est vide alors la pile d'ajout ne contient pas tous les éléments de file mais uniquement ceux enfilés à ce moment-là. Comme les fonctions ne renvoient rien, du fait du caractère mutable d'un tableau dynamique Python, il n'y a d'affectation que pour la création de la file.

Modularité

EXERCICE 1 ☞

1. Écrivez un module pour chaque structure de données des parties B, C, D et E du TP n° 2.
2. Écrivez l'interface commune à chacun de ces quatre modules.
3. Écrivez un module qui offre la fonction `genereDates` de la partie A du TP n° 2 et la fonction générique `contientDoublon` du cours.
4. Intégrez des levées d'exceptions ainsi que des vérifications de pré-conditions pour parer d'éventuels mésusages du module associé à la partie D.

   Vous créez ainsi un module `nsi_dates.py` qui sera exploitable pour la suite.

EXERCICE 2 ☞

On souhaite savoir combien il faut d'élèves en moyenne dans une école pour qu'un anniversaire soit fêté chaque jour de l'année civile.

Écrivez un module réalisant cette demande.

Vous pouvez prendre des initiatives!

EXERCICE 3 ☞

Proposez une solution pour vérifier expérimentalement le *paradoxe des anniversaires* tel qu'il a été énoncé en introduction du TP n° 2.

CORRIGÉ EXERCICE 1

Premiers modules

1. Il s'agit de spécifier chacune des fonctions `creerEnsemble`, `estPresent` et `ajouteElement` et d'écrire leurs codes avec la structure de données retenue.
   On met en œuvre la compétence **généraliser et abstraire**.

   On écrit ainsi cinq modules dont les codes sont disponibles dans le cours en ligne.

2. L'interface est la même car, pour chacun des modules, les trois fonctions réalisent la même chose.

Fonction	Description
`creerEnsemble()`	Crée et renvoie un ensemble de dates vide.
`estPresent(date, ens)`	Renvoie `True` si et seulement si la date `date` figure dans l'ensemble `ens`. Renvoie `False` sinon.
`ajouteElement(date, ens)`	Ajoute la date `date` dans l'ensemble `ens` préalablement créé.

3. Les fonctions plus spécifiques au problème étudié sont ici regroupées dans un même module.

   Le code est disponible dans le fichier `nsi_moduleDoublon.py`.

   On peut en écrire une interface.

Fonction	Description
`genereDates()`	Crée et renvoie un tableau de 23 dates calendaires aléatoirement comprise entre 1 et 366. Exemples : `echantillon = genereDates()`
`contientDoublon(dates)`	Renvoie `True` si et seulement si le tableau de dates contient au moins un doublon. Renvoie `False` sinon. Exemple : `contientDoublon(echantillon)`

   Il est prudent de tester l'ensemble produit et notamment chacun des modules d'ensemble.
   Le fichier `nsi_moduleDoublonTests.py` contient une adaptation des tests du TP n° 2 pour réaliser ce contrôle.

4. On recherche des dysfonctionnements possibles.
   Des exemples avec

```
>>> import nsi_modulePartieD as mod
>>> ens = mod.creerEnsemble()
>>> mod.ajouteElement(421, ens)  # provoque une erreur non compré...
    ...hensible depuis l'interface
>>> mod.ajouteElement(377, ens)  # ne provoque pas d'erreur mais ...
    ...inattendu
>>> mod.ajouteElement(-383, ens)  # ne provoque pas d'erreur mais...
    ... effet indésirable
```

   Des explications

   La structure de données choisie avec les 6 entiers encodés sur 64 *bits* permet de gérer $6 \times 64 = 384$ dates de 0 à 383.

   ① Pour `mod.ajouteElement(421, ens)`, $421 > 383$ d'où l'exception `IndexError` car l'expression `421 // 64` renvoie 6 et `len(ens)` est égale à 6.

   ② Pour `mod.ajouteElement(377, ens)`, la valeur 377 n'est pas une date mais elle ne génère pas d'erreur car $377 \leqslant 383$. On ne peut pas déterminer les éventuelles conséquences de cette saisie.

③ Pour `mod.ajouteElement(-383, ens)`, c'est plus fâcheux car `divmod(-383, 64)` renvoie la valeur `(-6,1)` or `ens[-6]` renvoie le premier entier du tableau et donc c'est la date 1 qui est ajoutée. Il s'agit là d'un effet indésirable car pour la personne utilisatrice cette date n'a pas été ajoutée or `mod.estPresent(1, ens)` renverra `True` là où l'on attend `false`.

Afin d'éviter ces problèmes, il convient de gérer ces comportements en interne du module et de renvoyer des messages d'erreurs compréhensibles à partir de la seule connaissance de l'interface.
Pour y parvenir, on mobilise des assertions et la gestion des exceptions en réfléchissant depuis le point de vue possible de la personne utilisatrice du module.
On réalise des **factorisations** d'une partie de ces codes de contrôle par des fonctions **privées** au sein du module.

Le module `nsi_dates.py` propose une série de vérifications des préconditions ou de tests du domaine de validité des valeurs saisies pour gérer en interne ces éventuels mésusages.
N'hésitez pas à tester les deux instructions

```
>>> import nsi_dates as dates
>>> help(dates)
```

pour observer le contenu affiché dans la console par rapport au statut **public** ou **privé** de chaque fonction composant ce module.

CORRIGÉ EXERCICE 2

Une autre exploitation des modules de structures de données

On conçoit un module spécifique pour ce problème en le décomposant.

- Une fonction pour calculer le nombre d'élèves, via leur date de naissance, à choisir aléatoirement avant d'avoir couvert les 366 dates possibles d'une année.
  L'algorithme s'organise autour d'une boucle `Tant que` dont la garde porte sur le nombre de dates calendaires couvertes.
  La théorie des probabilités et la **correction** de la fonction `randint` du module `random` nous assurent de la **terminaison** de cette boucle.
  Le code dépend du module `nsi_dates` pour gérer un ensemble de dates.
- Une fonction qui va répéter l'appel et renvoyer le nombre moyen d'élèves.
  L'algorithme s'organise autour d'une boucle `Pour`, afin de répéter l'expérience aléatoire, et d'une variable accumulateur pour cumuler le nombre de succès.
- Une fonction qui va répéter l'appel à la précédente et renvoyer les résultats dans un tableau.
  La valeur à renvoyer est calculée en compréhension.
- Une fonction qui va produire un histogramme à partir de la liste des résultats renvoyée par la fonction précédente afin de renseigner sur la répartition des résultats.
  L'instruction `help(plt.hist)` permet d'afficher la description de cette fonction.

Ce module dépend de plusieurs autres modules :

- `nsi_dates` pour gérer un ensemble de dates
- `random` pour générer des dates calendaires aléatoires
- `pyplot` de la bibliothèque `matplotlib` pour produire un histogramme.

Le code se trouve dans le fichier `nsi_moduleExo2Corrige.py`.

CORRIGÉ EXERCICE 3

Fin du problème initial

Deux démarches sont possibles :

① créer un nouveau module en important `nsi_moduleDoublon.py` pour utiliser les deux fonctions qu'il offre ;

② produire un module « intégré » en complétant `nsi_moduleDoublon.py` et dans ce cas les deux fonctions qu'il offre seront privatisées.

Deux fichiers différents contiennent les codes corrigés

① `nsi_moduleParadoxeV1.py` pour la première solution,

② `nsi_moduleParadoxeV2.py` pour la seconde.

Il nous faut répéter de nombreuses fois la séquence appel à `genereDates` puis à `contientDoublon`, compter le nombre de succès et observer la fluctuation de la fréquence des succès autour de 0,5 pour déterminer si elle est conforme à la théorie ou observer la stabilisation de la fréquence observée pour voir si elle se stabilise effectivement autour de 0,5.

Pour réaliser cet ensemble de tâches, il convient à nouveau de décomposer le problème comme dans l'exercice précédent.

Plusieurs fonctions supplémentaires sont alors nécessaires.

L'instruction `help(plt.plot` permet d'afficher la description de cette fonction et d'apprendre à s'en servir à partir de sa documentation.

Une interface s'impose enfin pour présenter les fonctions offertes et leur utilisation.

Fonction	Description
`frequenceSucces(n)`	Cette fonction renvoie la fréquence de succès lors de `n` répétitions de l'expérience aléatoire qui consiste à trouver un doublon parmi un choix aléatoire de 23 dates de naissance comprises entre 1 et 366.
`repeteFrequence(nbSeries, nbRepet)`	Cette fonction renvoie un tableau de taille `nbSeries` dont les valeurs sont les fréquences de succès observées lors de `nbRepet` répétitions de l'expérience aléatoire.
`frequenceCumuleeSucces(n)`	Cette fonction renvoie un tableau de taille `n` dont les valeurs sont égales à la fréquence cumulée de succès au fur et à mesure des n répétitions de l'expérience aléatoire.
`produitNuagePoints(listeFrequences)`	Cette fonction réalise un nuage de points représentant une liste de fréquences passée en argument. Elle nécessite le module `pyplot` de la bibliothèque `matplotlib`. L'affichage du résultat s'obtient par l'instruction `plt.show()` depuis le module.

La fonction `produitNuagePoints` permet de représenter chacun des deux tableaux renvoyés pour observer la fluctuation des fréquences des échantillons ou la stabilisation de la fréquence cumulée des échantillons.

Ce module propose aussi un code alternatif en cas d'exécution directe.

Sécurisation des communications

EXERCICE 1 ☞

On rappelle que le XOR, soit le « OU exclusif » en français, se note $\oplus$ comme opération logique et il est un opérateur informatique binaire qui pour les deux bits renvoie 0 s'ils ont la même valeur et 1 sinon.

Il peut être assimilé à l'addition modulo 2.

En pratique, cet opérateur est plutôt utilisé avec des octets et non bit par bit.

En langage Python, XOR peut s'appliquer sur des entiers, sur leurs encodages en fait, et il s'écrit ^ comme par exemple `3^2` qui renvoie `1`. En effet

$$3_{10} = 11_2 \text{ et } 2_{10} = 10_2 \text{ donc}$$

$$\begin{array}{rcc} & 1 & 1 \\ \oplus & 1 & 0 \\ \hline = & 0 & 1 \end{array}$$

or $01_2 = 1_{10}$ c'est pourquoi `3^2` renvoie la valeur entière `1`.

1. On donne ci-dessous les codes ASCII des premières lettres de l'alphabet en majuscule sur un octet ainsi que l'écriture décimale associée.

Lettre	A	B	C	D	E	F	G
Code	0100 0001	0100 0010	0100 0011	0100 0100	0100 0101	0100 0110	0100 0111
Valeur	65	66	67	68	69	70	71

   a. On considère la clef de chiffrement sur un octet 0000 0100.
   Déterminez le chiffrement de la lettre A avec cette clef.

   b. Chiffrez le chiffre de la lettre A avec la même clef.
   Que remarquez-vous ?

   c. Comment déchiffrer un message chiffré avec XOR ?

   d. Écrivez des instructions python réalisant les chiffrements précédents.

2. On souhaite implémenter le chiffrement XOR sur des chaînes de caractères en effectuant l'opération de chiffrement octet par octet comme à la question précédente.
   On rappelle que la fonction python `ord` renvoie l'écriture décimale de l'encodage ASCII du caractère passé en argument.
   Par exemple `ord('A')` renvoie `65`.
   Inversement, la fonction python `chr` renvoie le caractère ASCII à partir de l'écriture décimale de l'encodage binaire de ce caractère.
   Par exemple `chr(71)` renvoie `'G'`.

a. En supposant que la clef est une chaîne de caractères au moins aussi longue que le message à chiffrer, déterminez le code d'une fonction nommée `chiffrement_xor(message, clef...)` qui renvoie le message chiffré avec la clef en utilisant l'opérateur `XOR`.

b. Adaptez le code en considérant maintenant que la longueur de la clef peut être plus courte que celle du message à chiffrer.

EXERCICE 2 ☞

À l'aide de votre navigateur, consulter le site WEB de l'ANSSI appelé ssi.gouv.fr.

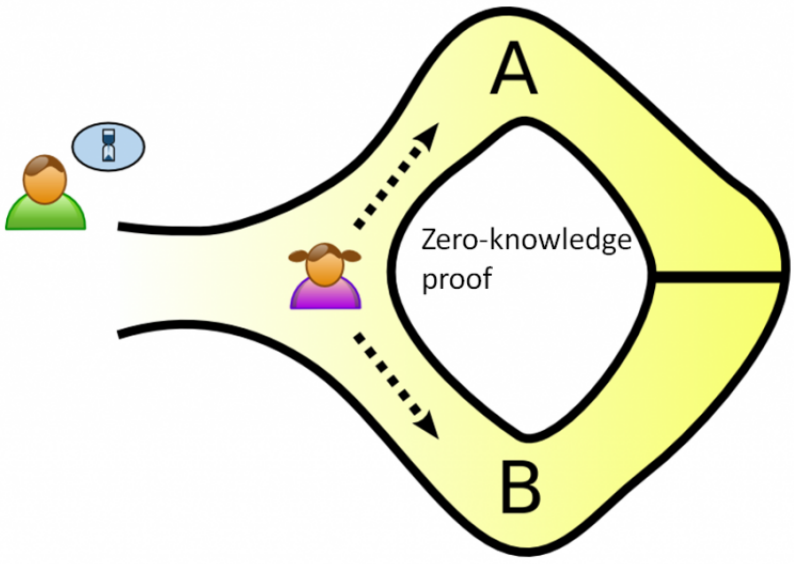
1. Que signifie le symbole cadenas qui apparaît à gauche de la barre d'adresse ?
2. En cliquant sur ce cadenas, vous avez la possibilité de faire afficher le certificat du serveur. Listez les éléments de ce certificat que vous parvenez à identifier.

EXERCICE 3 ☞

1. Saisissez dans la console l'instruction `help(pow)`
2. En exploitant la spécification ainsi obtenue, écrivez une série de fonctions renvoyant les éléments permettant à Alice et Bob de mettre en place un échange de clef de DIFFIE et HELLMAN.

EXERCICE 4 ☞

Alice propose à Bob d'aller visiter une grotte un peu particulière. Elle n'a qu'une issue et présente une bifurcation entre deux culs-de-sac fermés par une porte magique.



Alice prétend connaître la formule magique qui permet d'ouvrir cette porte mais elle ne peut pas la révéler à Bob sans déclencher un terrible sortilège.

Les deux amis cherchent donc un moyen de convaincre Bob qu'Alice connaît bien la formule.
Il mettent au point le protocole suivant.

① Bob attend à l'extérieur de la grotte sans regarder à l'intérieur. Alice entre dans la grotte, tire à Pile ou Face le choix entre les deux culs-de-sac puis se cache au fond de celui qu'elle a ainsi choisi au hasard.

② Bob pénètre dans la grotte jusqu'à la bifurcation. Il tire aussi à Pile ou Face entre les deux culs-de-sac et demande à Alice de sortir par celui qu'il a ainsi choisi au hasard.

③ Alice doit obligatoirement sortir par l'issue demandée par Bob.

1. Calculez la probabilité qu'Alice sorte par l'issue demandée par Bob sans connaître la formule magique.
2. Les deux amis recommencent n fois le protocole, avec des choix indépendants les uns des autres, pour $n \in \mathbf{N}^*$.
   Calculez la probabilité qu'Alice sorte à chaque fois par l'issue demandée par Bob sans connaître la formule magique.
3. Combien de fois faut-il itérer le protocole pour convaincre Bob qu'Alice connaît bien la formule magique ?

EXERCICE 5 ☞

1. Les clefs secrètes utilisées pour le protocole AES sont encodées sur un minimum de 128 bits.
   a. Quel est le nombre de clefs différentes possibles ?
   b. Donnez un équivalent décimal de cet ordre de grandeur.
2. Répondez aux deux mêmes questions pour les clefs retenues pour le protocole RSA encodées sur un minimum de 2 048 bits.

EXERCICE 6 ☞

Le module `hashlib` offre notamment plusieurs fonctions de hachage.
On donne ci-dessous le code d'une fonction qui exploite deux de ses fonctionnalités.

```
1 import hashlib
2
3 def hache_sha1(chaine : str):
4   """Renvoie le haché d'une chaîne de caractères selon la méthode
5       de hachage sha1.
6   """
7   octets = chaine.encode()  # en objet de type byte
8   hache = hashlib.sha1(octets)  # objet haché
9   return hache.hexdigest()  # renvoi en hexadécimal
```

1. Examen de cette fonction
   a. Testez cette fonction après en avoir recopié le code.
   b. Quelle est la longueur de sortie en octets ? en bits ?
   c. Les longueurs des hachages sont-elles différentes pour des chaînes de longueurs différentes ?
   d. Exécutez la fonction deux fois de suite avec le même argument. Qu'observez-vous ?
   e. Même question en modifiant un seul des caractères de l'argument.
2. Les fonctions de hachage sont particulièrement utilisées pour stocker les hachés des mots de passe. Pour cela on introduit un « sel », qui est un élément aléatoire et qui est ajouté au mot de passe avant le hachage.
   Pour contrôler un mot de passe saisi, le sel et le haché sont récupérés dans une base de données et il est alors possible de réaliser la comparaison avec la saisie utilisateur.
   a. Écrivez le code d'une fonction `hache_mdpSel(sel, mdp)` qui renvoie le sel et le haché avec cette méthode.
   b. Comment l'utiliser pour hacher ? pour vérifier une saisie de mot de passe ?

3. Petit test de résistance

a. En exploitant la fonction `time` du module `time`, écrivez le code d'une fonction de signature `test_hache(n : int) -> float:` qui renvoie la durée nécessaire pour hacher `n` fois le même mot de passe.

b. Proposez une utilisation de la fonction `test_hache` pour tester la résistance de cette méthode de hachage à une attaque par force brute.

CORRIGÉ EXERCICE 1

Chiffrement symétrique avec `XOR`

1. Pour bien comprendre le « ou exclusif ».
   a. On pose l'opération comme avec l'exemple.

$$\begin{array}{cccccccc} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ \oplus & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ = & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{array}$$

   Ainsi, la lettre A est chiffrée par la lettre E.

   b. On réitère l'opération.

$$\begin{array}{cccccccc} & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ \oplus & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ = & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{array}$$

   Ainsi, la lettre E est chiffrée par la lettre A, autrement dit l'opération de chiffrement permet le déchiffrement.

   c. Il suffit d'appliquer à nouveau le « ou exclusif » avec la même clef pour déchiffrer le message.
   On dit que cette opération est **involutive**.
   Cette propriété rend le `XOR` très simple à exploiter et aussi très rapide car l'opération bit à bit est implémentée dans les circuits du processeur.

   d. On mobilise l'opérateur python `^` avec les valeurs décimales sachant que celle de la clef est 4.

```
>>> 65^4
69
>>> 69^4
65
```

2. Une implémentation.
   a. On vérifie la pré-condition puis on mobilise correctement les fonctions natives python.
   On peut tester cette fonction avec les exemples précédents.

   b. On va utiliser en plus l'opérateur modulo % sur la valeur de la variable de boucle pour ne pas déborder la chaîne `clef`.

En s'appuyant sur la méthode `encode` de la classe `str` (voir ici), sur la classe `bytes`, qui permet de manipuler des octets (voir là), et notamment sa méthode `decode`, il est possible d'adapter cette fonction à un message encodé en `utf-8`.

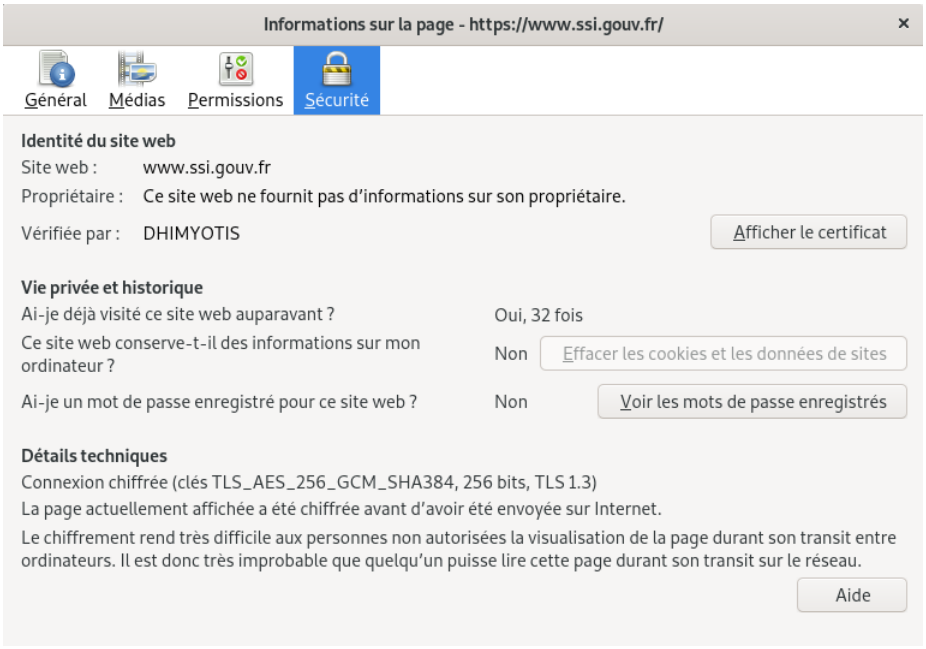
On applique la méthode `encode()` aux chaînes de caractères à traiter et la méthode `decode()` à l'objet `byte` renvoyé.

```
Console
Python 3.9.2 (/bin/python3)
>>> %Run nsi_secureComEx1.py
>>> mess = "Voilà un réel écrit pour utf-8."
>>> clef = "ù*µ£ê74!"
>>> chiffre = chiffrement_xor_ter(mess.encode(), clef.encode())
>>> chiffre
b'\x95\xd6C\xaevb\x83\xb6\xc4\x17F\xe2j\xdcF\xe2vk\xc0\xb1\xc3C\x14Q\xac\xccX\xe2\xc0
\xb6\xc5\xee\x92\x19'
>>> type(chiffre)
<class 'bytes'>
>>> chiffrement_xor_ter(chiffre, clef.encode()).decode()
'Voilà un réel écrit pour utf-8.'
>>>
```

CORRIGÉ EXERCICE 2

Examen d'un certificat

1. Le symbole cadenas qui apparaît à gauche de la barre d'adresse illustre la sécurisation des échanges de données avec le serveur qui héberge ce site.
   En cliquant sur ce symbole, on parvient à afficher la fenêtre ci-dessous.



   La rubrique **détails techniques** affiche un item **connexion chiffrée** avec

   `TLS_AES_256_GCM_SHA384, 256 bits, TLS 1.3`

   qui signifie que cette connexion sécurisée utilise le protocole TLS dans sa version 1.3, le chiffrement symétrique est assuré par le protocole AES avec une clef de 256 bits. Le complément GCM signifie que l'intégrité des données est assurée par ce système de chiffrement, dit intègre. Le sigle SHA384 désigne la fonction de hachage utilisée.
2. Voici une liste d'éléments du certificat qu'il est possible d'identifier.
   - nom de l'autorité : DHIMYOTIS
   - période de validité : du 20/09/2023 au 19/09/2024
   - clef publique du chiffrement asymétrique : protocole RSA avec une taille de 2 048 bits.
   - l'algorithme de signature combine la fonction de hachage SHA256 avec le protocole RSA.
   - clef publique du client et de l'autorité de certification
   - le niveau de certification OV

Pour aller plus loin, vous pouvez lire ce guide publié par l'ANSSI.

CORRIGÉ EXERCICE 3

Exponentiation modulaire

1. On saisit `help(pow)` et on obtient :

```
Python 3.9.2 (/bin/python3)
>>> help(pow)
Help on built-in function pow in module builtins:

pow(base, exp, mod=None)
    Equivalent to base**exp with 2 arguments or base**exp % mod with 3 arguments

    Some types, such as ints, are able to use a more efficient algorithm when
    invoked using the three argument form.
```

On relève ainsi qu'il est possible de calculer la puissance modulaire d'un entier avec cette fonction native même s'il existe des algorithmes plus performants dans ce cas particulier.

```
>>> pow(2, 4, 5)
1
```

car $2^4 = 16$ et $16 = 3 \times 5 + 1$.

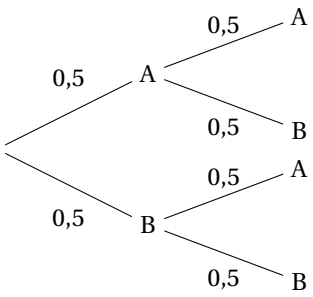
2. Alice et Bob doivent choisir un grand nombre premier p et un nombre entier a compris entre 1 et $p-1$.
Ces deux valeurs doivent donc être passées en arguments des fonctions.
Alice comme Bob doivent aussi choisir chacun un nombre secret, dont la valeur passera aussi en argument.
Il est possible d'écrire une seule fonction générique de calcul pour encapsuler le principe.
Le programme principal de mise en scène peut être réduit à des saisies supposées conformes et un renvoi.

Voir les codes dans le cours en ligne.

CORRIGÉ EXERCICE 4

Un peu de calcul de probabilités

1. On peut représenter la situation à l'aide d'un arbre de probabilité.



Le premier choix est celui d'Alice, le second celui de Bob. Comme Bob choisit aussi au hasard entre les deux culs-de-sac, désignés par A et B sur cet arbre, sans information sur le choix d'Alice, la probabilité de chacun de ses choix est aussi égale à 0,5.

La probabilité qu'Alice sorte par l'issue demandée par Bob sans connaître la formule magique se calcule donc par

$$p = \mathrm{P}(\text{Alice a choisi A et Bob aussi}) + \mathrm{P}(\text{Alice a choisi B et Bob aussi}) = 0{,}5 \times 0{,}5 + 0{,}5 \times 0{,}5 = 0{,}5$$

Elle est donc égale à 0,5.
On désigne par S l'évènement : « Alice sort par l'issue désignée par Bob sans connaître la formule magique ».
L'évènement contraire, noté $\overline{\mathrm{S}}$, signifie donc que soit Alice est sortie par l'issue désignée par Bob car elle connaît la formule magique, soit elle est sortie par l'autre issue faute de connaître cette formule.

2. Il s'agit d'exprimer la probabilité qu'Alice réalise l'évènement S à chaque fois sur les n répétitions indépendantes, cette probabilité est donc égale à $0{,}5^n$.

Les élèves qui ont choisi la spécialité mathématique auront reconnu la loi binomiale.

3. On peut calculer la première valeur de n pour laquelle $0{,}5^n < s$ où s est un seuil de probabilité fixé.
Soit on connaît la fonction logarithme népérien, alors la réponse est immédiate, sinon il suffit de concevoir un algorithme de seuil et de le coder.
On produit alors un tableau des valeurs de la variable `n` pour une liste de valeurs choisies pour la variable `s`. Ainsi, par une définition en compréhension, on peut compléter le fichier contenant la fonction précédente avec le programme principal suivant :

```
# programme principal
if __name__ == "__main__":
  print([seuil(0.5, s) for s in [1e-1, 1e-2, 1e-3, 1e-4, 1e-5,\
   1e-6, 1e-7, 1e-8]])
```

On trouve alors les réponses :

```
[4, 7, 10, 14, 17, 20, 24, 27]
```

Ainsi par exemple, dès 10 répétitions de l'expérience, la probabilité de réussite d'Alice sans connaître la formule magique est inférieure à $1 \cdot 10^{-3}$.

En conclusion, un nombre suffisant de répétitions de l'expérience permet à Bob d'être presque sûr (au sens des probabilités, voir cette page wikipedia par exemple) qu'Alice connaît la formule magique sans que cette dernière n'ait eu à la divulguer.

Voir les codes dans le cours en ligne.

CORRIGÉ EXERCICE 5

Un peu de calcul

1. Taille des clefs du protocole AES.

a. Chacun des 128 bits ne peut prendre que 2 valeurs qui sont 0 ou 1 donc le nombre de clefs différentes possibles est

$$\underbrace{2 \times 2 \times \cdots \times 2}_{\text{128 fois}} = 2^{128}$$

soit l'entier 340 282 366 920 938 463 463 374 607 431 768 211 456 qui est bien difficile à lire.

b. On peut compter le nombre de chiffres ou trans-typer la valeur en chaîne de caractères et renvoyer sa taille.

```
>>> len(str(2**128))
39
```

Un équivalent décimal est donc $3{,}4 \cdot 10^{38}$.

2. Pour les clefs retenues pour le protocole RSA, on trouve de la même manière

$$\underbrace{2 \times 2 \times \cdots \times 2}_{\text{2048 fois}} = 2^{2048}$$

soit un entier s'écrivant avec 617 chiffres pour un ordre de grandeur décimal de $3{,}2 \cdot 10^{616}$.

Vous trouverez avec ces liens :

- un bon complément sur l'importance de la taille des clefs ici
- un outil de calcul de la force d'un mot de passe sur le site de l'ANSSI.

CORRIGÉ EXERCICE 6

Fonction de hachage

1. Examen de cette fonction
   a. On exécute par exemple `hache_sha1("Mot de passe")` qui renvoie une chaîne de caractères désignant des chiffres hexadécimaux.
   b. Avec `len(hache_sha1("Mot de passe"))` la valeur renvoyée est 40. Cette chaîne est donc composée de 40 chiffres hexadécimaux, chacun d'eux étant codé sur 4 bits on en déduit que la longueur de la sortie est
      - $40 \div 2 = 20$, soit 20 octets ;
      - $40 \times 4 = 160$, soit 160 bits.
      - Avec `len(hache_sha1("Mot de passe beaucoup plus long"))` on peut observer que la valeur renvoyée est à nouveau 40 et émettre la conjecture que les longueurs des hachés sont constantes même si les chaînes de caractères passées en argument sont de tailles distinctes.
      - Si on exécute deux fois de suite la fonction avec la même valeur pour l'argument, alors on observe que la valeur du haché est la même.

        Par contre, la modification d'un seul caractère conduit à un haché complètement différent.
2. Introduction d'un « sel ».
   a. On écrit les codes de deux fonctions, l'une qui génère un sel aléatoire de taille choisie en argument et l'autre qui est `hache_mdpSel`.

      Pour générer aléatoirement un sel, on peut créer un alphabet automatiquement avec la fonction `chr` déjà rencontrée en choisissant bien les valeurs décimales des caractères sélectionnés à partir d'une table ASCII puis choisir au hasard un caractère avec remise dans cet alphabet, par exemple avec la fonction `choice` du module `random`.

      Il suffit ensuite d'adapter le code de la fonction `hache_sha1` en intégrant le sel.
   b. Pour hacher un mot de passe avec un sel de taille 6, on peut alors écrire l'instruction suivante.

```
>>> sel, h = hache_mdpSel(cree_sel(6), "C'est mon mot de passe")
```

      La valeur du sel et le haché du mot de passe étant stockés dans une base de données, on peut contrôler la saisie d'un mot de passe en
      - hachant le mot de passe saisi avec le sel associé au client
      - puis en comparant ce haché avec celui figurant dans la base de données.

```
>>> mdpSaisi = "C mon mot de passe"
>>> sel, hSaisi = hache_mdpSel(sel, mdpSaisi)
>>> h == hSaisi
False
```

3. Petit test de résistance
   a. On effectue une prise de l'heure avant une boucle `Pour` simulant n répétitions du hachage du même mot de passe puis une autre après et on renvoie la différence entre les deux.

      La fonction renvoie une durée en seconde.

b. On va mesurer la durée nécessaire sur un ordinateur grand public pour hacher un grand nombre de fois le même mot de passe.

En affinant, on constate qu'il est possible de hacher environ 1,5 million de mots de passe en 1 s.

Vérifier un mot de passe est donc beaucoup trop rapide avec cette méthode de hachage et cela la rend vulnérable à une attaque par la force brute.

Un mot de passe à hacher par sha1 ne peut être encodé que sur un maximum de 2^{64} bits. Comme un caractère ASCII est encodé sur 8 bits, le nombre maximum de caractères qu'un mot de passe peut comporter est donc égal à $2^{64} \div 2^3 = 2^{61}$.

Prenons comme exemple un mot de passe composé de 12 caractères choisis parmi les 94 présents dans l'alphabet étendu. Il y a donc 94^{12} mots de passe possibles, soit environ $4{,}76 \cdot 10^{23}$.

En divisant par 1 500 000, on obtient la durée approximativement nécessaire pour les tester tous : $3{,}17 \cdot 10^{17}$ s. Ce qui donne $3{,}67 \cdot 10^{12}$ j soit environ $1 \cdot 10^{10}$ a.

Un ordinateur personnel seul reste donc insuffisant pour réaliser ce type d'attaque mais comme elle peut être distribuée en parallèle, il suffit de l'organiser depuis un grand nombre de machines.

- Vous pourrez étudier avec profit la documentation python officielle relative au module hashlib.
- Pour connaître les principes de l'algorithme de hachage sha1 ainsi que les failles qui ont été prouvées par des équipes de recherche, vous pourrez consulter cette page Wikipedia.

Récursivité

EXERCICE 1 ☞

1. Comparez les deux fonctions suivantes codées en Python, en anticipant leurs exécutions et leurs résultats.

```
def figure1(n):
  """n est un entier naturel"""
  if n > 0:
    print(n * "*")
    figure1(n - 1)
  return None
```

```
def figure2(n):
  """n est un entier naturel"""
  if n > 0:
    figure2(n - 1)
    print(n * "*")
  return None
```

2. Codez en Python une fonction itérative nommée `figure3` (respectivement `figure4`) qui donne le même résultat que la fonction `figure1` (respectivement `figure2`).
3. On modifie la fonction nommée `figure2` de la façon suivante :

```
def figure5(n):
  """n est un entier naturel"""
  if n > 0:
    figure5(n - 1)
    print(n * "*")
    figure5(n - 1)
  return None
```

   a. Représentez un arbre des appels récursifs pour l'exécution `figure5(3)`.

   b. Exprimez le nombre d'appels récursifs en fonction de la valeur de l'argument `n` dans le cas général.

   c. Déterminez l'affichage produit par l'exécution `figure5(3)`.

EXERCICE 2 ☞

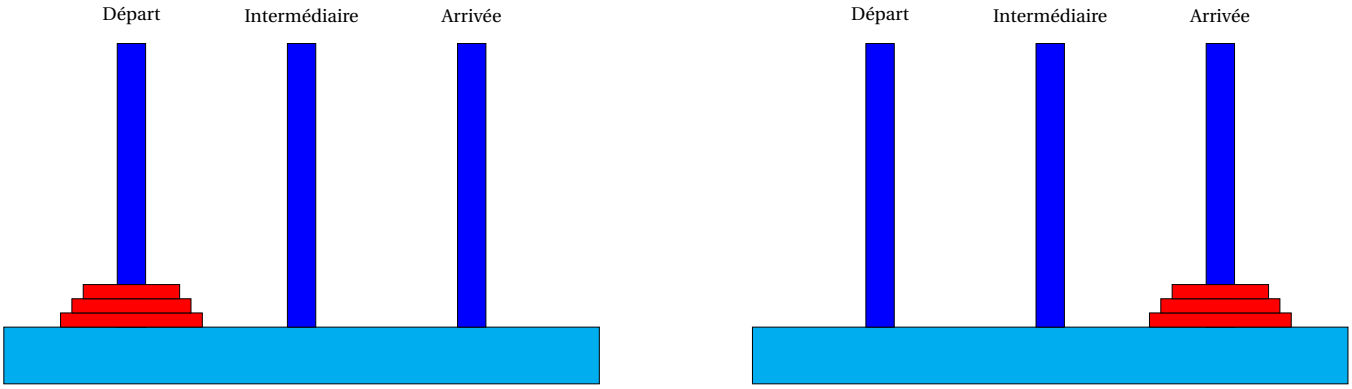
Dans le jeu des tours de Hanoï, n disques percés en leur centre, où n désigne un nombre entier naturel, sont empilés sur un poteau par taille décroissante sur un poteau par taille décroissante.

Le but du jeu est d'amener ces disques sur un autre poteau et dans la même configuration en utilisant uniquement un troisième poteau et en respectant les deux règles suivantes.

① on ne peut déplacer qu'un seul disque à la fois, celui situé au sommet d'une pile, en le plaçant au sommet d'une autre pile ;

② on ne peut pas déposer un disque sur un disque de taille inférieure.

On peut schématiser, dans le cas $n = 3$, la situation initiale et le but comme sur les deux figures ci-dessous.



Il s'agit d'établir un algorithme récursif de résolution de ce jeu pour une quantité n de disques quelconque.

1. Dans le cas où $n = 2$ et en numérotant les deux disques ① et ② du plus petit au plus grand, proposez une solution en donnant les états successifs des trois poteaux.
2. Dans le cas $n = 3$ et en numérotant ①, ② et ② les disques du plus petit au plus grand.
   a. Déterminez une solution détaillée en sept mouvements dans ce cas.
   b. Exprimez plus succinctement, en trois temps, une résolution récursive dans ce cas.
   c. Que devez-vous supposer avec l'expression succincte de la résolution ? Est-ce raisonnable ?
3. Exprimez une solution récursive dans le cas général et vérifiez la avec les cas connus.
4. Écrivez l'algorithme d'une fonction récursive qui renvoie le nombre de déplacements $d(n)$ réalisés en fonction du nombre n de disques initiaux.

EXERCICE 3

On considère la suite numérique u définie pour tout nombre entier naturel n par

$$u(0) = 0 \quad ; \quad u(1) = 1 \quad \text{et} \quad u(n) = u(n-2) + u(n-1) \quad \text{pour } n \geqslant 2$$

1. Calculez à la main les valeurs de $u(2)$, $u(3)$, $u(4)$ et $u(5)$.
2. Déterminez un algorithme récursif de calcul du terme $u(n)$ de cette suite.
3. Représentez l'arbre des appels pour l'évaluation de $u(5)$ avec cet algorithme.
4. Imaginez une solution au problème soulevé par l'arbre des appels précédent.

EXERCICE 4

On appelle logarithme entier d'un nombre réel x supérieur ou égal à 1, le nombre de divisions successives par 2 à appliquer à x pour obtenir un résultat inférieur ou égal à 1.

Par exemple, pour $x = 60\,000$ on trouve 16 car $2^{16} = 65\,536$ et $2^{15} = 32\,768$. On note elog(60 000) = 16.

Écrivez un algorithme récursif de calcul du logarithme entier d'un nombre réel supérieur ou égal à 1.

CORRIGÉ EXERCICE 1

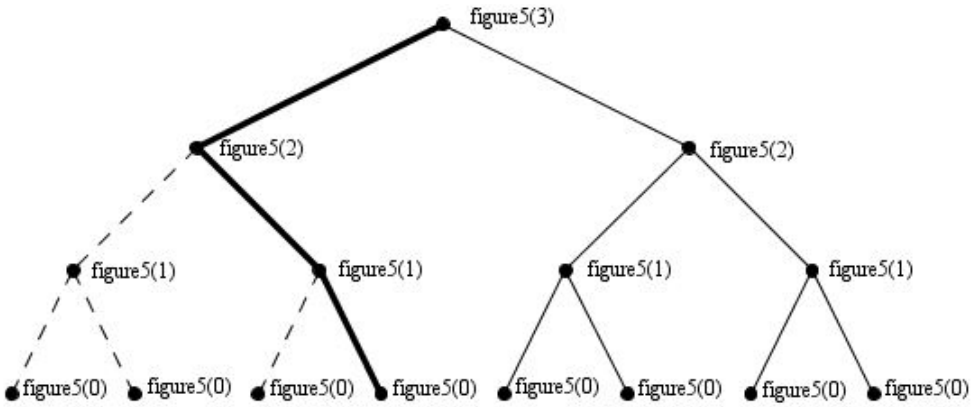
Analyser un code récursif

1. **Comparaison des deux fonctions récursives :**
   La différence entre ces deux fonctions est la position de `print(n*"*")` : soit avant, soit après l'appel récursif.
   La fonction `figure1` affiche une ligne avec n astérisques, puis une ligne avec $n-1$ astérisques, etc... jusqu'à une ligne avec 1 astérisque.
   La fonction `figure2` effectue tous les appels récursifs avant de commencer l'affichage des lignes d'astérisques.
   Ainsi l'affichage commence pour la valeur 1 et donc elle affiche une ligne avec 1 astérisque, puis une ligne avec 2 astérisques, etc... jusqu'à une ligne avec n astérisques. en remontant la pile des appels récursifs.
2. **Deux fonctions itératives :**

```
def figure3(n):
  """n est un entier naturel"""
  for k in range(n, 0, -1):
    # arrêt à 1 astérisque
    print(k * "*")
  return None
```

```
def figure4(n):
  """n est un entier naturel"""
  for k in range(1, n + 1):
    # départ à 1 astérisque
    print(k * "*")
  return None
```

3. En complément de l'arbre et de la pile des appels récursifs écrits au tableau, voici un arbre des appels récursifs descendant qui illustre une exécution en cours de la fonction pour laquelle les branches en trait fin correspondent aux appels qui n'ont pas encore été effectués, les branches en trait épais correspondent aux appels en cours d'exécution et les branches en pointillés correspondent aux appels dont l'exécution est terminée.



Le contenu de la pile d'exécution récursive à ce moment précis est donc :

figure5(0)
figure5(1)
figure5(2)
figure5(3)

et l'affichage réalisé à ce moment de l'exécution est le suivant :

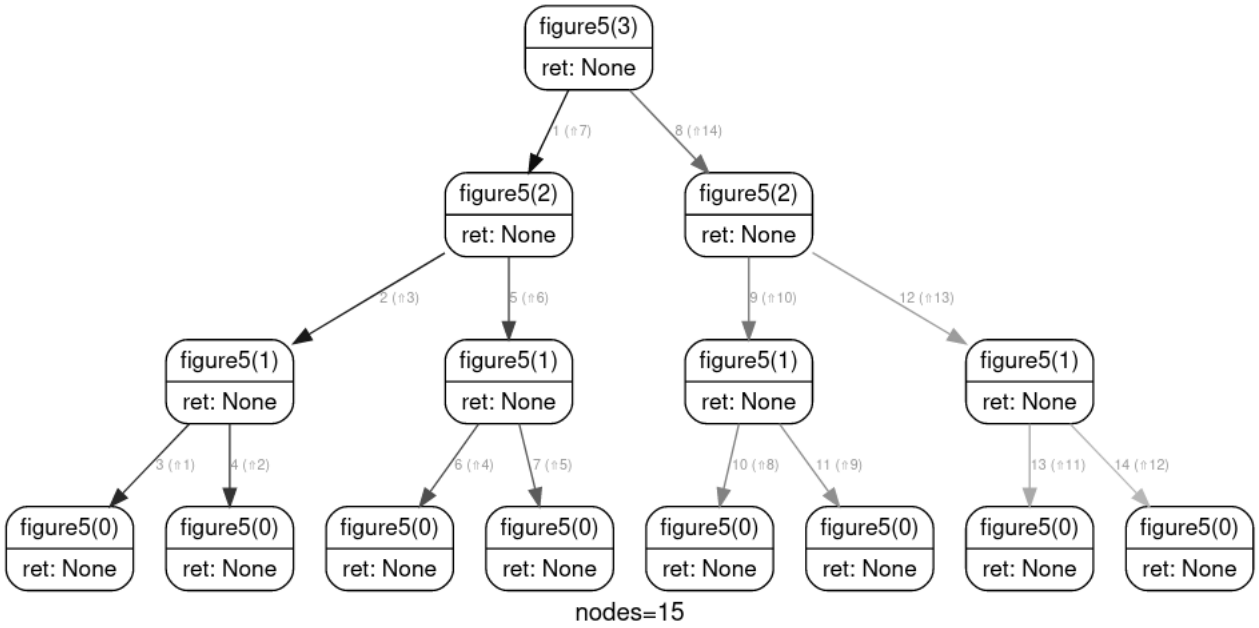
```
*
**
*
```

En effet, lors de l'exécution, l'affichage de n astérisques est réalisé juste après la fin de la première exécution de `figure5(n-1)`, donc quand on remonte dans l'arbre des appels et en sachant qu'un appel de `figure5(0)` ne provoque aucun affichage.

Le résultat final est donc :

```
*
**
*
***
*
**
*
```

L'arbre final des appels récursifs et des remontées est complètement symétrique.



Dans cette représentation vous pouvez lire les numéros des remontées.

Pour l'appel principal `figure5(n)` où la valeur de l'argument `n` est un entier naturel non nul, les appels récursifs effectués se dénombrent ainsi

- 1 appel principal `figure5(n)`.
- Cet appel déclenche 2 appels récursifs avec la valeur $n-1$.
- Chacun de ces 2 appels déclenche 2 appels récursifs avec la valeur $n-2$, soit $2\times 2 = 2^2$ appels récursifs.
- Chacun de ces 2^2 appels déclenche 2 appels récursifs avec la valeur $n-3$, soit $2^2\times 2 = 2^3$ appels récursifs.
- …
- Chacun des 2^{n-2} déclenche 2 appels récursifs avec la valeur $n-(n-1)=1$, soit $2^{n-2}\times 2 = 2^{n-1}$ appels récursifs de `figure5(1)`.
- Chacun des 2^{n-1} déclenche 2 appels récursifs avec la valeur $n-n=0$, soit $2^{n-1}\times 2 = 2^n$ appels récursifs de `figure5(0)`.

Le nombre total d'appels récursifs pour une valeur entière positive n est donc égal à

$$1+2+2^2+\cdots+2^n = 2^{n+1}-1$$

On peut vérifier avec $n=3$ où $2^{3+1}-1 = 2^4-1 = 16-1 = 15$.

Preuve informatique de la formule de la somme

La somme $1+2+2^2+\cdots+2^n$ est la valeur décimale du plus grand nombre entier naturel qui peut être encodé sur $n+1$ *bits* car

$$1+2+2^2+\cdots+2^n = 1\times 2^n + 1\times 2^{n-1} + \cdots + 1\times 2 + 1\times 2^0$$

puisque $2^0 = 1$.

La valeur de cet entier naturel est donc égale à celle de l'entier suivant que l'on encoderait sur $n+2$ *bits* moins 1 or l'encodage de cet entier suivant est décrit par

$$1\times 2^{n+1} + 0\times 2^n + 0\times 2^{n-1} + \cdots + 0\times 2^1 + 0\times 2^0$$

donc sa valeur décimale est égale à celle de 2^{n+1}.

On a ainsi prouvé l'égalité $1+2+2^2+\cdots+2^n = 2^{n+1}-1$.

Pour terminer, on note que les 2^n appels récursifs de `figure5(0)` ne réalisent rien et qu'il serait donc opportun de s'en défaire.
Il suffit pour cela de modifier la condition de poursuite en `n > 1` et de traiter le cas de base pour la valeur 1.

```
def figure6(n):
  """n est un entier naturel"""
  if n > 1:
    figure6(n - 1)
    print(n * "*")
    figure6(n - 1)
  else:
    print("*")
  return None
```

CORRIGÉ EXERCICE 2

Les tours de Hanoï

1. Cas $n = 2$ en numérotant ① et ② les deux disques du plus petit au plus grand.

Mouvements	Départ	Intermédiaire	Arrivée
Étape 0	② & ①	rien	rien
Étape 1	②	①	rien
Étape 2	rien	①	②
Étape 3	rien	rien	② & ①

2. Cas $n = 3$ en numérotant ①, ② et ③ les trois disques du plus petit au plus grand.

a. Une solution détaillée en sept mouvements.

Mouvements	Départ	Intermédiaire	Arrivée
Étape 0	③ & ② & ①	rien	rien
Étape 1	③ & ②	rien	①
Étape 2	③	②	①
Étape 3	③	② & ①	rien
Étape 4	rien	② & ①	③
Étape 5	①	②	③
Étape 6	①	rien	③ & ②
Étape 7	rien	rien	③ & ② & ①

On peut remarquer qu'à l'étape 3, si l'on fait abstraction du disque ③ au poteau de départ, la situation est analogue à celle de l'étape 0 du cas $n = 2$ où le poteau intermédiaire jouerait le rôle de celui de départ et le poteau de départ celui du poteau intermédiaire.

Les étapes 5, 6 et 7 du cas $n = 3$ sont donc les étapes 1, 2 et 3 du cas $n = 2$ en changeant les rôles de ces poteaux.

b. Pour exprimer en trois temps une résolution récursive dans ce cas, on va justement supposer que l'on sait résoudre le cas $n = 2$.

Les trois temps se décrivent alors comme dans ce tableau.

Mouvements	Départ	Intermédiaire	Arrivée
Temps 0	③ & ② & ①	rien	rien
Temps 1	③	② & ①	rien
Temps 2	rien	② & ①	③
Temps 3	rien	rien	③ & ② & ①

Les passages du temps 0 au temps 1 et du temps 2 au temps 3 sont effectués par appels récursifs.

c. On doit supposer que l'on sait traiter ces deux appels récursifs, ce qui est vrai avec

- pour le premier, le poteau arrivée qui joue le rôle d'intermédiaire et celui d'intermédiaire qui joue le rôle d'arrivée ;
- pour le second, le poteau départ qui joue le rôle d'intermédiaire et celui d'intermédiaire qui joue celui de départ.

Cette supposition ne fait donc pas obstacle à la résolution du problème.

3. Dans le cas général avec $n \in \mathbf{N}$, il suffit d'exprimer l'algorithme en supposant que l'on sait résoudre le problème pour $n-1$ disques, ce qui est vrai car pour $n = 0$ où il n'y a rien à faire et pour $n = 1$ où il suffit de déplacer le disque au poteau d'arrivée.

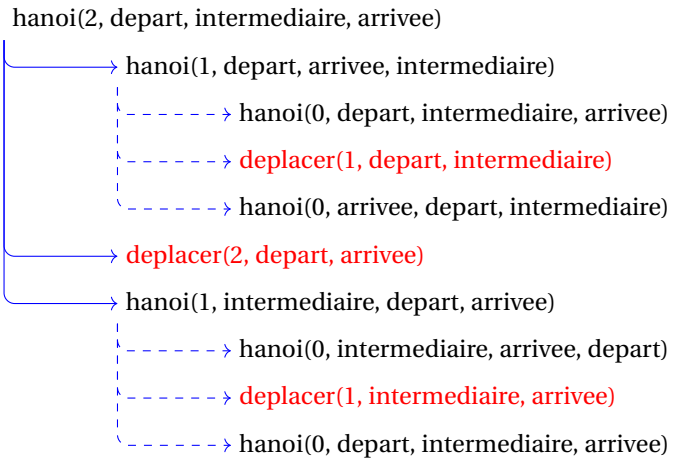
On a ainsi identifié les appels récursifs et la condition d'arrêt avec le traitement de chaque cas de base.

La fonction récursive va comporter 4 arguments, la valeur de n et chacun des trois poteaux.

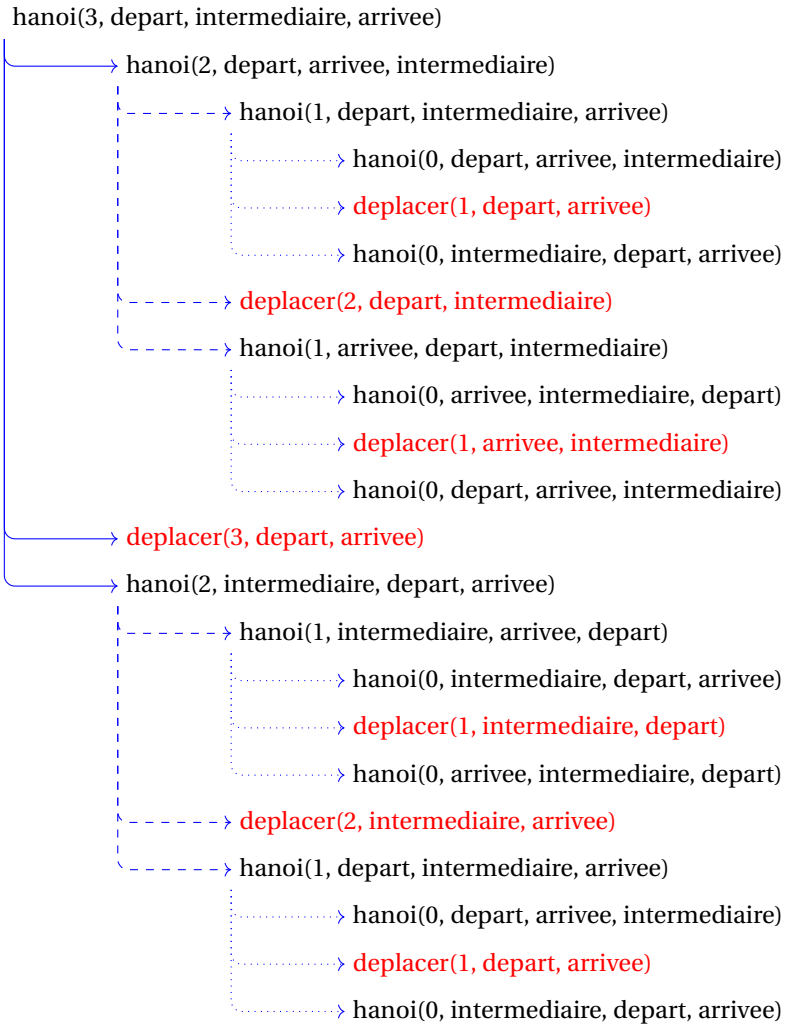
```
1 fonction hanoi
2 Input : n, depart, intermediaire, arrivee  // entier, ...
3 Si n > 0 Alors
4   hanoi(n - 1, depart, arrivee, intermediaire)
5   deplacer(n, depart, arrivee)
6   hanoi(n - 1, intermediaire, depart, arrivee)
7 // sous-entendu Sinon rien
8 Fin Si
9 Output : rien
```

On vérifie la validité de cet algorithme avec les deux cas connus.

- Pour $n = 0$, la condition $n > 0$ est fausse donc il ne se passe rien et c'est le comportement attendu.
- Pour $n = 1$, le premier appel récursif `hanoi(0, depart, arrivee, intermediaire)` sera sans effet avec la valeur 0 pour le premier argument et il en sera de même du second appel récursif. Seule l'action `deplacer(1, depart, arrivee)` sera réalisée et c'est ce qui est attendu.
- Pour $n = 2$, détaillons un arbre des appels récursifs.



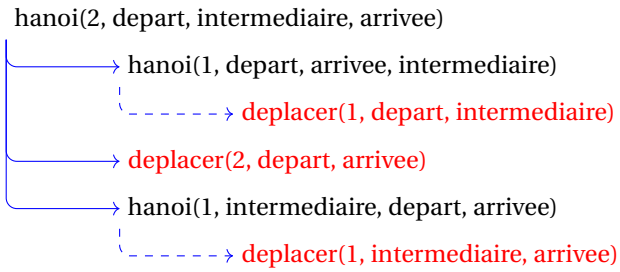
- Pour $n = 3$, on procède de la même manière.



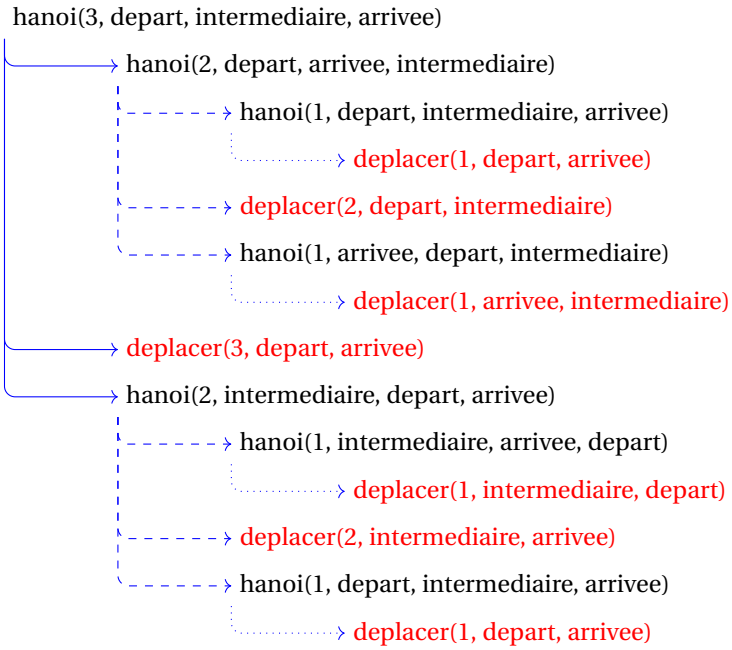
Les nombreux appels avec la valeur `0` pour l'argument `n` ne réalisent aucune action. Il paraît donc avantageux de les éliminer en modifiant la condition d'arrêt en `Si n > 1 Alors` et en complétant avec le traitement du cas de base pour $n = 1$ dans l'alternative `Sinon Si`. On obtient ainsi l'algorithme modifié :

```
fonction hanoi
Input : n, depart, intermediaire, arrivee  // entier, ...
Si n > 1 Alors
  hanoi(n - 1, depart, arrivee, intermediaire)
  deplacer(n, depart, arrivee)
  hanoi(n - 1, intermediaire, depart, arrivee)
Sinon Si n = 1 Alors
  deplacer(n, depart, arrivee)
// sous entendu Sinon rien
Fin Si
Output : rien
```

La vérification dans les cas $n = 0$ et $n = 1$ est immédiate. Pour les cas $n = 2$ on trouve :



et pour $n = 3$ on obtient :



Chaque appel est productif. C'est un point sur lequel il faut être vigilant car la mémoire de la machine peut être très sollicitée par la profondeur des appels récursifs.

4. On revient au raisonnement récursif. Pour déplacer n disques du départ à l'arrivée, il suffit de déplacer les $n-1$ du dessus du départ à l'intermédiaire puis 1 disque, celui du fond, du départ à l'arrivée et enfin les $n-1$ disques de l'intermédiaire à l'arrivée.

On obtient ainsi la relation $d(n) = d(n-1) + 1 + d(n-1)$ soit $d(n) = 2 \times d(n-1) + 1$.

Pour le cas de base avec $n = 1$, on effectue 1 déplacement et donc $d(1) = 1$ et pour celui avec $n = 0$ on en effectue 0 soit $d(0) = 0$.

Comme $1 = 2 \times 0 + 1$ l'égalité $d(n) = 2 \times d(n-1) + 1$ est vraie aussi pour $n = 1$.

Un algorithme d'une fonction récursive qui renvoie la valeur de $d(n)$ pour un entier naturel quelconque n s'écrit donc

```
fonction nbreDeplacementsHanoi
Input : n  // un nombre entier naturel non nul
Si n = 0 Alors
  Renvoyer 0
Sinon
  Renvoyer 2 × nbreDeplacementsHanoi(n - 1) + 1
```

L'optimisation du code comme vue en cours et sur les exemples relève de la programmation.

CORRIGÉ EXERCICE 3

Suite de FIBONACCI

On considère la suite numérique u définie pour tout nombre entier naturel n par

$$u(0) = 0 \quad ; \quad u(1) = 1 \quad \text{et} \quad u(n) = u(n-2) + u(n-1) \quad \text{pour } n \geqslant 2$$

1. Pour $n = 2$, on substitue puis on calcule $u(2) = u(0) + u(1) = 0 + 1 = 1$.
De même pour $n = 3$, $n = 4$ et $n = 5$ et on trouve

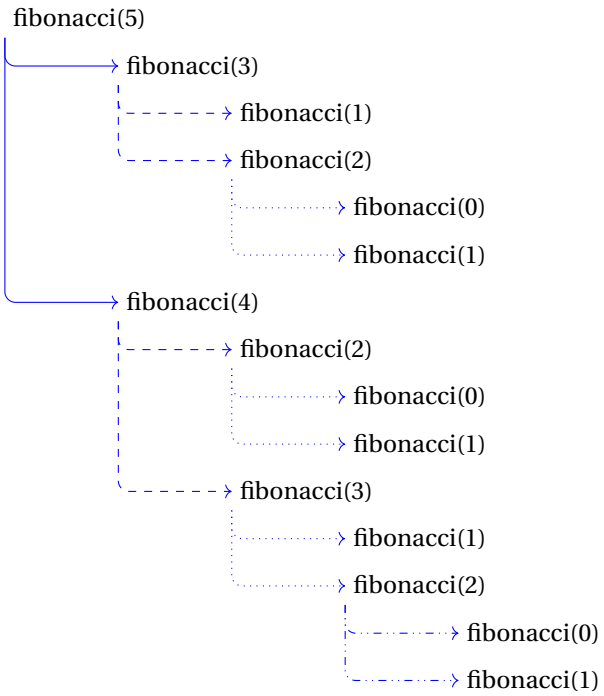
$$u(3) = u(1) + u(2) = 1 + 1 = 2$$
$$u(4) = u(2) + u(3) = 1 + 2 = 3$$
$$u(5) = u(3) + u(4) = 2 + 3 = 5$$

2. Dans cet algorithme, deux cas de base sont à considérer et on doit réaliser un double appel récursif pour calculer le terme dans les autres cas.
Un algorithme très explicite peut alors s'écrire :

```
fonction fibonacci
Input : n  // entier
Si n > 1 Alors
  Renvoyer fibonacci(n - 2) + fibonacci(n - 1)
Sinon Si n = 1 Alors
  Renvoyer 1
Sinon
  Renvoyer 0
Fin Si
```

Les deux dernières conditions peuvent être regroupées dans la seule alternative `Sinon` dont le bloc d'instructions se réduit à `Renvoyer n` mais il faut alors assurer la pré-condition $n \geqslant 0$ au préalable.

3. Représentons l'arbre des appels comme dans le cours pour l'exécution avec la valeur 5.



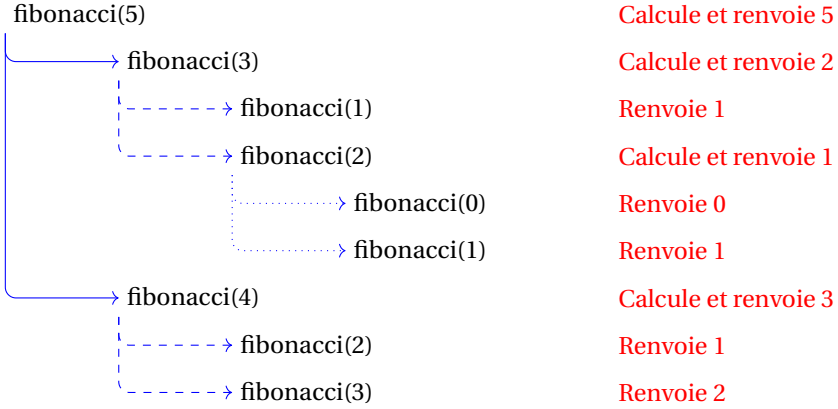
On remarque la répétition des mêmes appels qui conduit à calculer plusieurs fois une même valeur. C'est un facteur de ralentissement de l'exécution et de consommation inutile de ressources.

4. Résoudre ce problème demande de trouver une solution algorithmique qui permette de ne pas calculer plusieurs fois une même valeur.

- Une solution passe par la mémorisation des valeurs déjà calculées dans un dictionnaire dont les clefs sont les valeurs de n et les valeurs celles de $u(n)$.
  Ce dictionnaire peut alors constituer une variable globale car sa valeur est mutable.
  Cette méthode est s'appelle la **mémoïsation**.

```
1 dico ⟵ {0:0, 1:1}  // dictionnaire des valeurs calculées
2 fonction fibonacci
3 Input : n, dico  // entier, dictionnaire
4 Si n > 1 Et n n'est pas une clef de dico Alors
5   dico[n] ⟵ fibonacci(n - 2) + fibonacci(n - 1)
6 Fin Si
7 Renvoyer dico[n]
```

Observons les appels pour l'exécution avec la valeur 5 en distinguant les renvois directs de valeurs figurant déjà dans le dictionnaire des calculs effectués avant renvoi après ajout de la valeur associée à la clef idoine dans le dictionnaire.



Avec cette solution, la mémoire est moins sollicitée mais on mémorise quand même toutes les valeurs d'indices inférieurs à n, ce qui peut faire beaucoup si n est très grand.

- Une seconde solution repose sur la seule mémorisation des valeurs des deux termes précédents pour un calcul du terme suivant ce qui nécessite de mettre à jour les valeurs de ces deux termes à chaque appel récursif.
  À la place d'un dictionnaire on n'utilise donc que deux variables, désignées ici par `a` et `b`, dont les valeurs sont mises à jour dans l'appel récursif.
  Cette méthode illustre le **paradigme fonctionnel**, pas de boucle et pas d'affectation.

```
a ⟵ 0  // valeur du premier terme
b ⟵ 1 // valeur du deuxième terme
fonction fibonacci
Input : n, a, b  // trois entiers naturels
Si n = 0 Alors
  Renvoyer a
Sinon Si n = 1 Alors
  Renvoyer b
Sinon
  Renvoyer fibonacci(n - 1, b, a + b)
Fin Si
```

Dans cet algorithme, la valeur de l'argument `n` représente plus un compte à rebours du nombre d'appels à effectuer plutôt que l'indice du terme.
Il présente aussi l'avantage de ne plus comporter qu'un simple appel récursif à la place d'un double.
Observons les appels pour l'exécution avec la valeur 5.

fibonacci(5, 0, 1)
→ fibonacci(4, 1, 1)
→ fibonacci(3, 1, 2)
→ fibonacci(2, 2, 3)
→ fibonacci(1, 3, 5) Renvoie 5

Avec cette solution, il n'y a pas de calcul lors du dépilage des appels, la récursivité est **terminale**.

CORRIGÉ EXERCICE 4

Logarithme entier

On appelle logarithme entier d'un nombre réel x supérieur ou égal à 1, le nombre de divisions successives par 2 nécessaires pour obtenir un résultat inférieur ou égal à 1.

Avec la pré-condition $x \geqslant 1$, on écrit un premier algorithme.

```
fonction logEntier
Input : x  // un flottant supérieur ou égal à 1
Si x > 1 Alors
  Renvoyer logEntier(x / 2) + 1
Sinon
  Renvoyer 0
Fin Si
```

Avec cet algorithme, on accumule en mémoire les appels récursifs avant d'effectuer les additions au fur et à mesure du dépilage avec la valeur renvoyée par l'appel récursif qui suivait.

On va à nouveau rendre la **récursivité terminale** en ajoutant un argument optionnel, qui est ici une variable compteur, dont la valeur est initialisée à 0.

```
1 cpt ⟵ 0  // valeur pour x = 1
2 fonction elog
3 Input : x , cpt // un flottant supérieur ou égal à 1, un entier
4 Si x > 1 Alors
5   Renvoyer elog(x / 2, cpt + 1)
6 Sinon
7   Renvoyer cpt
8 Fin Si
```

On a ainsi à nouveau réduit l'usage de la mémoire et accélérer le traitement.

Cette fonction renvoie en fait la partie entière du nombre réel $\log_2(x)$ pour tout nombre réel $x \geqslant 1$.

Diviser pour régner

EXERCICE 1 ☞

1. Complétez pour commencer le code de la fonction nommée `_separe` dont on rappelle ci-dessous la signature et la spécification.

```
def _separe(tab : list, val) -> Tuple[list, list]:
    """Cette fonction sépare les éléments du tableau tab en deux
       sous-tableaux, celui des éléments dont la valeur est
       inférieure ou égale à val d'une part et les autres
       d'autre part.
    """
```

   Vérifiez qu'elle passe les tests puis proposez d'autres tests.

2. Déterminez alors le code d'une fonction nommée `triRapide` qui renvoie un tableau dont les éléments sont triés en ordre croissant en mettant en œuvre l'algorithme du tri rapide.

```
def triRapide(tab : list) -> list:
    """Cette fonction renvoie un tableau composé des éléments
       de tab triés en ordre croissant.
       Elle met en œuvre l'algorithme de tri rapide.
       Elle est récursive.
    """
```

3. Utilisez le fichier de tests pour tester votre code avec `pytest`.

EXERCICE 2 ☞

1. Pour chaque couple de deux sous-tableaux triés, exécutez l'étape de fusion en cherchant à la concevoir sous forme algorithmique.
   a. Avec | 1 | 3 | 5 | 8 | 9 | et | 2 | 3 | 4 | 7 |.
   b. Avec | 1 | 3 | 5 | 8 | et | 3 | 4 | 10 |.
2. Complétez le code de la fonction nommée `_fusion` puis celui de la fonction `triFusion`.
3. Utilisez le fichier de tests pour tester votre code avec `pytest`.

EXERCICE 3 ☞

La recherche dichotomique d'une valeur dans un tableau trié croissant a été vue en classe de première. On y revient dans cet exercice en observant qu'une stratégie dichotomique est une stratégie « diviser pour régner » particulière.

On dispose d'un tableau de valeurs de même type dont les éléments sont triés. On souhaite déterminer si une valeur, externe et toujours du même type, est présente dans le tableau. Dans ce cas, on peut mettre en œuvre une recherche dichotomique en appliquant la stratégie suivante.

Stratégie dichotomique

① On considère l'élément du milieu du tableau (où presque ...).

② On compare sa valeur avec celle qui est à rechercher.

③ On recommence avec la partie gauche ou la partie droite du tableau selon le résultat obtenu à la comparaison.

Un algorithme itératif peut s'écrire ainsi.

Recherche dichotomique dans un tableau trié

```
Input : val, tab  // une valeur, un tableau
n ⟵ taille de tab
g ⟵ 0
d ⟵ n - 1
Tant que g ⩽ d Faire
  m ⟵ quotient de (g + d) divisé par 2
  Si tab[m] > val Alors
    d ⟵ m - 1
  Sinon Si tab[m] < val Alors
    g ⟵ m + 1
  Sinon
    Renvoyer Vrai // val est trouvée
  Fin Si
Fin Tant que
Output : Faux // val non présente dans tab
```

1. Appliquez cet algorithme pour rechercher la valeur 7 dans le tableau

0	1	1	2	3	5	8	13	21

2. Quel est le comportement de cet algorithme si le tableau passé en argument effectif est vide ? S'il ne contient qu'un seul élément ?

3. Codez cet algorithme en complétant la fonction `rechercheDichotomique`.
On rappelle que le quotient de la division euclidienne s'obtient en Python par l'opération `//`.

4. Utilisez le fichier de tests pour tester votre code avec `pytest`.

5. Complétez le code de la version récursive avec la fonction `rechercheDichotomique_rec`.

CORRIGÉ EXERCICE 1

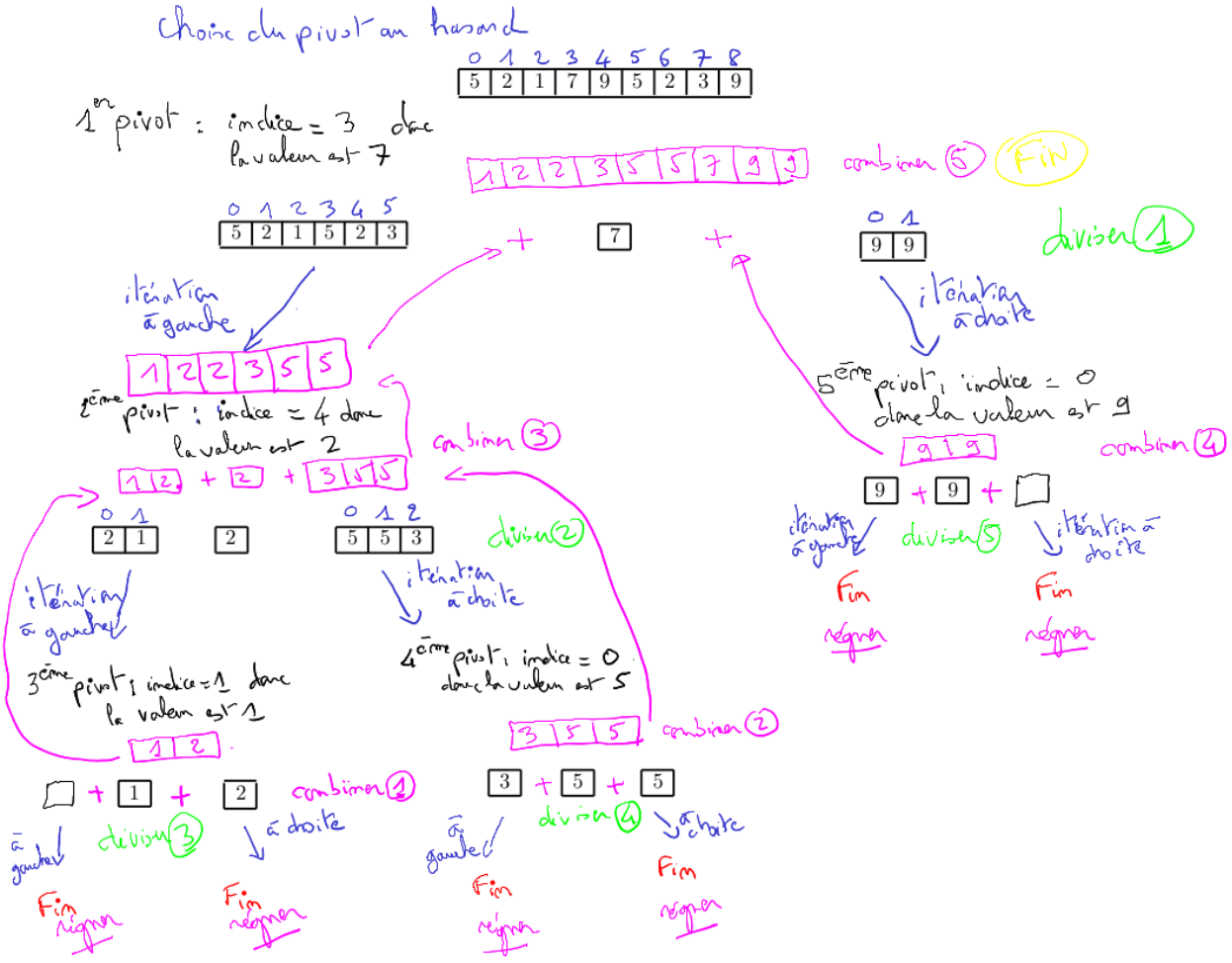
Codage du tri rapide

Avant de commencer, voici un déroulé possible de l'algorithme avec le tableau de nombres entiers suivant

5	2	1	7	9	5	2	3	9

en choisissant l'indice du pivot au hasard.

La descente est décrite en bleu avec le choix du pivot en noir, la remontée l'est en violet. On identifie ensuite les occurrences de chacune des trois étapes : diviser, régner, combiner.



On choisit l'indice du pivot au hasard puis on applique l'étape **diviser**. On itère ensuite d'abord à gauche tant que la taille du tableau de gauche après division n'est pas inférieure ou égale à 1. C'est alors l'étape **régner**. On réalise la même chose sur le tableau de droite issu de la division précédente. Les tableaux renvoyés à l'issue de l'étape **régner** sont assemblés autour du pivot correspondant, c'est l'étape **combiner**.

On peut observer sur cet exemple ce que pourront être les appels récursifs et la structure de l'algorithme : **diviser** puis **régner** et **combiner** les deux tableaux renvoyés autour du pivot.

1. On peut parcourir la tableau `tab` par indice ou par valeur. Ici on le fait par valeur, le code est moins chargé et très lisible.

   En sortie de cette fonction, on sait que toutes les valeurs des éléments de `tabGauche` sont inférieures ou égales à celle de `val` et celles des éléments de `tabDroit` sont supérieures.

   Pour les tests, il faut faire attention à bien attendre un `tuple` comme valeur renvoyée et à placer les valeurs des éléments de `tab` comme elles se sont présentées dans le parcours. Les deux sous-tableaux ne sont pas triés par cette fonction.

2. On code une fonction récursive dont la condition d'arrêt porte sur la taille du tableau, c'est l'étape **régner**, et qui appelle la fonction nommée `_separe` pour **diviser** après avoir choisi une valeur comme **pivot** et ôté celle-ci du tableau.
   Enfin, après les appels récursifs, l'étape **combiner** se traduit par « l'assemblage » des trois tableaux triés, gauche - pivot - droit.
   Attention alors à bien placer la valeur de la variable `pivot` comme un tableau composé d'un seul élément.

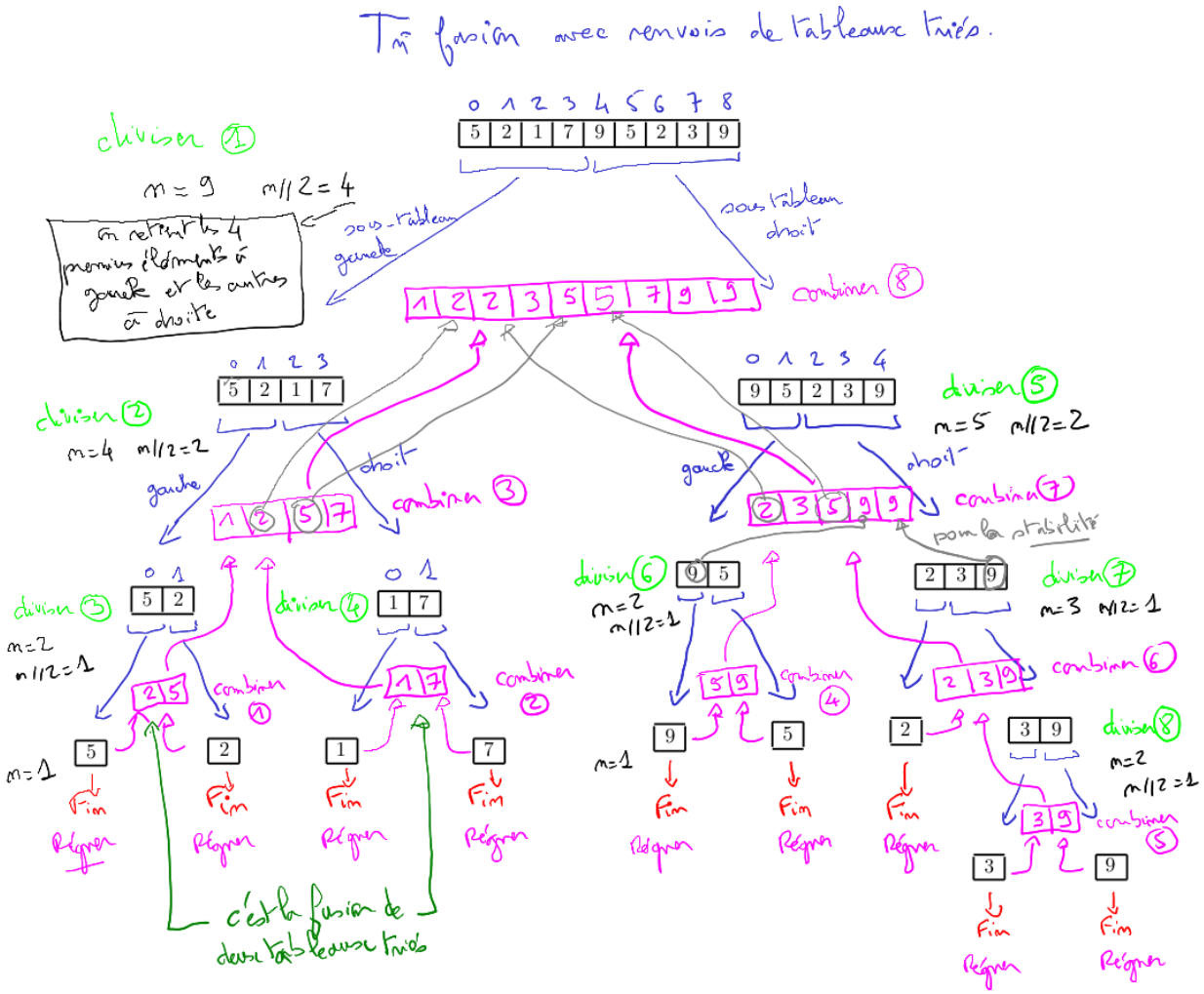
CORRIGÉ EXERCICE 2

Codage du tri fusion

Avant de commencer, voici un déroulé possible de l'algorithme avec le tableau de nombres entiers suivant

5	2	1	7	9	5	2	3	9

La descente est décrite en bleu, la remontée l'est en violet. On identifie ensuite les occurrences de chacune des trois étapes : diviser, régner, combiner. Lors de la division, le quotient de la division euclidienne de la taille par 2 donne le nombre d'éléments à placer dans le sous-tableau gauche, sinon l'algorithme ne termine pas. Lors de la fusion, en cas d'égalité entre deux valeurs, on insère en premier celle du sous-tableau gauche afin de garantir la propriété de ***tri stable****.*



On calcule la séparation puis on applique l'étape **diviser**. On itère tant que la taille du tableau est supérieure à 1. C'est alors l'étape **régner**. Il reste ensuite à fusionner les tableaux triés en remontant. C'est l'étape **combiner**.
On observe que l'étape **combiner** est plus complexe à réaliser qu'avec l'algorithme précédent.
Il faut donc se pencher avec rigueur sur cette étape avant de coder l'ensemble.

1. Le plus important est de parvenir à traduire algorithmiquement la démarche et à relever que la comparaison de deux éléments avec $\leqslant$ permettra de réaliser un tri stable.
2. Plusieurs solutions sont possibles pour coder l'algorithme de fusion, certaines mobilisent plus de boucles. Un algorithme classique pour la fusion s'exprime comme suit.

```
Input : tab1, tab2  // deux tableaux triés
// traitement
n1 ⟵ taille de tab1
n2 ⟵ taille de tab2
i1 ⟵ 0  // indice de parcours de tab1
i2 ⟵ 0  // indice de parcours de tab2
result ⟵ tableau vide  // pour recueillir les valeurs triées
Tant que i1 < n1 Et i2 < n2 Faire  // pour ne pas déborder
  Si tab1[i1] ⩽ tab2[i2] Alors
    Ajouter tab1[i1] à result
    i1 ⟵ i1 + 1
  Sinon
    Ajouter tab2[i2] à result
    i2 ⟵ i2 + 1
  Fin Si
Fin Tant que  // tab1 ou tab2 est entièrement parcouru
// Cas où tab1 n'est pas entièrement parcouru (i1 < n1)
Pour i allant de i1 à n1 Faire
  Ajouter tab1[i] à result
Fin Pour
// Cas où tab2 n'est pas entièrement parcouru (i2 < n2)
Pour i allant de i2 à n2 Faire
  Ajouter tab2[i] à result
Fin Pour
// Une seule des boucles est réalisée car i1 ⩾ n1 ou i2 ⩾ n2
Output : result
```

L'algorithme du tri fusion n'est pas trop compliqué à concevoir.

```
fonction mergeSort_rec
Input : tab  // un tableau de valeurs comparables par <
// traitement
Si taille de tab ⩽ 1 Alors
  Renvoyer tab  // régner
Sinon
  // diviser
  m ⟵ quotient de la taille de tab par 2
  tabGauche ⟵ éléments de tab d'indice 0 à m - 1
  tabDroit ⟵ éléments de tab d'indice supérieur ou égal à m
  // régner récursivement
  tabGaucheTrie ⟵ mergeSort_rec(tabGauche)
  tabDroitTrie ⟵ mergeSort_rec(tabDroit)
  // combiner
  Renvoyer fusion de tabGaucheTrie et de tabDroitTrie
Output : un nouveau tableau trié
```

Les tests sont les mêmes que ceux choisis pour le tri rapide.

Preuve de l'algorithme

Pour rappel, la preuve d'un algorithme s'établit en deux parties : la **terminaison**, on prouve que l'algorithme se termine quelle que soit la valeur de l'entrée, et la **correction partielle**, on prouve que l'algorithme réalise ce qui est attendu s'il se termine.

Terminaison

Il faut prouver que l'algorithme de fusion se termine ainsi que l'algorithme récursif.

1. La **fusion**

   On introduit un **variant** pour la boucle `Tant que` avec la quantité $(n_1 - i_1) + (n_2 - i_2)$ en montrant que sa valeur est entière, positive et qu'elle décroît à chaque tour de boucle.

   Entière À l'initialisation les valeurs des quatre variables `n1`, `n2`, `i1` et `i2` sont des nombres entiers naturels.
   Seules les deux variables `i1` et `i2` voient leur valeur modifiée et c'est par ajout du nombre entier 1.
   On a montré : la quantité $(n_1 - i_1) + (n_2 - i_2)$ est toujours un nombre entier.

   Positive Initialement les valeurs de `n1` et `n2` sont positives ou nulles et celles de `i1` et `i2` sont nulles ainsi la somme $(n_1 - i_1) + (n_2 - i_2)$ est positive ou nulle.
   La garde de la boucle `Tant que` assure que $n_1 - i_1 > 0$ et $n_2 - i_2 > 0$, ainsi avant d'entrer dans la boucle la somme des deux est positive.
   La boucle s'arrête si $n_1 - i_1 = 0$ ou $n_2 - i_2 = 0$ et la somme est alors positive ou nulle.
   On a montré : la quantité $(n_1 - i_1) + (n_2 - i_2)$ est toujours un nombre positif ou nul.

   Stricte décroissance À chaque tour de boucle, soit la valeur de `i1` augmente de 1 soit c'est celle de `i2` ainsi, soit la différence $n_1 - i1$ diminue de 1 soit c'est $n_2 - i2$ et dans tous les cas la somme des deux diminue de 1.
   On a montré : la quantité $(n_1 - i_1) + (n_2 - i_2)$ est strictement décroissante.

   En conclusion, la quantité $(n_1 - i_1) + (n_2 - i_2)$ est un nombre entier positif ou nul strictement décroissant à chaque tour de boucle donc il ne peut y avoir qu'un nombre fini de tours, au plus $n_1 + n_2$, et la boucle se termine.
   Les deux boucles `Pour` se terminent nécessairement car le nombre de tours est fini.

2. La **fonction récursive**

   On va montrer qu'il y a un nombre fini d'appels récursifs et d'appels à la fusion. On note n la taille du tableau à trier.
   Les appels récursifs portent sur des tableaux dont les tailles sont strictement décroissantes.
   En effet, les tailles successives des tableaux sont de l'ordre de $n/2$, $n/4$, $n/8$, etc.
   Cette suite strictement décroissante de nombres positifs s'écrit aussi $\frac{n}{1}, \frac{n}{2^1}, \frac{n}{2^2}, \frac{n}{2^3}$, etc.
   Les divisions cessent donc pour la première valeur entière de k telle que $\frac{n}{2^k} \leqslant 1$.
   On nomme ainsi k le plus petit entier naturel tel que $2^k \geqslant n$ et on a donc $k = \text{elog}(n)$.
   D'autre part, le premier appel récursif en génère 2 autres qui, chacun en génère 2 autres, etc.
   Le nombre total d'appels récursifs est donc égal à $1+2+2^2+\cdots+2^k = 2^{k+1}-1$ soit de l'ordre de $2^{k+1} = 2 \times 2^k$ c'est-à-dire $2n$.
   Le nombre de fusions est lui aussi fini car il y a :

   - 1 fusion de 2 tableaux de tailles environ $n/2$;
   - 2 fusions de 2 tableaux de tailles environ $n/4$;
   - $4 = 2^2$ fusions de 2 tableaux de tailles environ $n/8$;
   - ...
   - 2^{k-1} fusions de 2 tableaux de tailles environ $n/2^k$ (0 ou 1).

Ce qui donne un nombre total de fusions égal à $1+2+2^2+\cdots+2^{k-1}=2^k-1$ qui est donc de l'ordre de n.

En conclusion, les appels récursifs sont en nombre fini et se terminent, les fusions aussi, donc l'algorithme se termine.

Correction partielle

La correction est essentiellement celle de la **fusion**.

Pour la démontrer, on introduit un **invariant** c'est-à-dire une propriété qui est vraie avant la boucle et qui reste vraie à chaque tour de boucle.

Cette propriété sera donc vraie à la sortie de la boucle, c'est pourquoi on la formule selon le résultat attendu. Ici un tableau trié.

Propriété invariante : pour toute valeur de i_1 et de i_2, le tableau `result[0:i1+i2-1]` est trié.

Avant la boucle Comme les valeurs de `i1` et `i2` sont nulles alors le tableau `result[0:-1]` est vide et il est donc trié.

Tour de boucle On suppose que `result[0:i1+i2-1]` est trié pour les valeurs actuelles de `i1` et `i2` et aussi que $i1 < n1$ et $i2 < n2$.

Deux cas sont seulement possibles.

① Soit `tab1[i1]`$\leqslant$ `tab2[i2]` et dans ce cas `tab1[i1]` va être ajouté à `result` or

- `tab1[i1]`$\geqslant$ `tab1[i1-1]` car `tab1` est trié
- `tab1[i1]`$>$ `tab2[i2-1]` car `tab2[i2-1]` a été inséré avant sous la condition `tab1[i1...-1]`$>$ `tab2[i2-1]` ou `tab1[i1]`$>$ `tab2[i2-1]`.

Ainsi `tab1[i1]` est supérieur ou égal à l'élément inséré au tour précédent.

② `tab1[i1]`$>$ `tab2[i2]` et dans ce cas `tab2[i2]` va être ajouté à `result` or

- `tab2[i2]`$\geqslant$ `tab2[i2-1]` car `tab2` est trié
- `tab2[i2]`$>$ `tab1[i1-1]` car `tab1[i1-1]` a été inséré avant sous la condition `tab2[i2...-1]`$>$ `tab1[i1-1]` ou `tab2[i2]`$>$ `tab1[i1-1]`.

Ainsi `tab2[i2]` est supérieur ou égal à l'élément inséré au tour précédent.

Enfin, la valeur de $i1+i2$ augmente de 1 et donc à l'issue d'un tour de boucle, le tableau `result[0:i1...+i2+1-1]` soit `result[0:i1+i2]` est trié.

Conclusion À la sortie de la boucle la propriété est vraie ainsi que $i_1 = n_1$ ou $i_2 = n_2$ et donc

① soit `result[0:n1+i2-1]` est trié et $i_1 = n_1$;

② soit `result[n2+i1-1]` est trié et $i_2 = n_2$.

Dans le cas ① la deuxième boucle `Pour` va maintenir la propriété invariante car `tab2` est trié et dans le cas ② c'est la première car `tab1` est trié.

En conclusion, à l'issue de l'algorithme `result[0:n1+n2-1]` est trié, ce qui est le résultat attendu.

Calcul de la complexité

Par renvoi de nouveaux tableaux, la **complexité en mémoire** est de l'ordre de $2n$ où n est la taille du tableau à trier.

Estimons ensuite la **complexité en temps**.

- La fusion de deux tableaux triés de tailles respectives m et p est de l'ordre de $m+p$ car elle réalise essentiellement un seul parcours de chacun des deux tableaux.
- La division en deux tableaux d'un tableau de taille m génère m opérations de copie des m éléments dans les deux sous-tableaux. Elle est donc de l'ordre de m.

Pour la suite on note toujours $k = \text{elog}(n)$.

Comptons alors le nombre de divisions et d'opérations de copie qu'elles nécessitent :

- 1 division du tableau de taille n soit n opérations de copie ;
- 2 divisions de 2 tableaux de taille $n/2$ soit n opérations de copie ;
- $2^2 = 4$ divisions de 4 tableaux de taille $n/4$ soit n opérations de copie ;
- ...
- 2^{k-1} divisions de 2^{k-1} tableaux de taille $n/2^{k-1}$ soit n opérations de copie ;

ce qui donne un total de k opérations de division ayant chacune un coût total de l'ordre de n avec les copies d'éléments, soit une complexité de l'ordre de $k \times n$.

Comptons enfin le nombre de fusions et d'opérations qu'elles nécessitent :

- 1 fusion de 2 tableaux de taille $n/2$ soit un coût de $n/2 + n/2 = n$;
- 2 fusions de 2 tableaux de taille $n/4$ soit un coût de $(n/4 + n/4) \times 2 = n$;
- $2^2 = 4$ fusions de 2 tableaux de taille $n/8$ soit un coût de $(n/8 + n/8) \times 4 = n$;
- 2^{k-1} fusions de 2 tableaux de taille $n/2^k$ soit un coût de $(n/2^k + n/2^k) \times 2^{k-1} = n/2 + n/2 = n$;

ce qui donne un total de k opérations de fusion ayant chacune un coût d'ordre n, soit une complexité de l'ordre de $k \times n$.

Par additivité, on obtient une complexité en temps de l'ordre de $2 \times n \times k$ or $k \approx \log_2(n)$ soit une complexité d'ordre $n \times \log_2(n)$ ou $\mathcal{O}(n \ln n)$.

On démontre qu'il s'agit de la complexité optimale pour un tri.

CORRIGÉ EXERCICE 3

La recherche dichotomique

1. On recherche la valeur 7 dans le tableau | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |.
On complète un tableau d'état des variables en exécutant l'algorithme ligne à ligne.

Ligne	État des variables
l2, l3 & l4	`n` 9 `g` 0 `d` 8
l5	`g<= d` Vrai
l6	`m` 4
l7	`tab[m]` 3 `tab[m] > val` Faux
l9	`tab[m]` 3 `tab[m] < val` Vrai
l10	`g` 5
l5	`g<= d` Vrai
l6	`m` 6
l7	`tab[m]` 8 `tab[m] > val` Vrai
l8	`d` 5
l5	`g<= d` Vrai
l6	`m` 5
l7	`tab[m]` 5 `tab[m] > val` Faux
l9	`tab[m]` 5 `tab[m] < val` Vrai
l10	`g` 6
l5	`g<= d` Faux
l15	`Renvoyer Faux`

Il peut être pertinent de pointer les indices g et d sur le tableau. de nombres pour bien comprendre l'algorithme.

2. Si le tableau passé en argument effectif est vide alors

$$\texttt{n}\boxed{0} \quad \texttt{g}\boxed{0} \quad \texttt{d}\boxed{-1} \quad \texttt{g}\leqslant\texttt{d}\boxed{\text{Faux}}$$

On n'entre pas dans la boucle `Tant que` et l'algorithme renvoie `Faux` quelle que soit la valeur de `val`.

C'est le comportement attendu.

Si le tableau passé en argument ne contient qu'un seul élément alors

$$\texttt{n}\boxed{1} \quad \texttt{g}\boxed{0} \quad \texttt{d}\boxed{0} \quad \texttt{g}\leqslant\texttt{d}\boxed{\text{Vrai}}$$

Si la valeur de `val` n'est pas égale à celle de `tab[0]` alors $\texttt{d}\boxed{-1}$ ou $\texttt{g}\boxed{1}$ selon le résultat de la comparaison entre `tab[0]` et `val` ; au tour suivant l'algorithme renverra alors `Faux` et sinon, en ligne 12, l'algorithme renverra `Vrai`.

C'est aussi le comportement attendu.

3. Le codage de cette version itérative ne pose pas de problème particulier.

4. On teste avec l'exemple de la question 1, avec un tableau vide, un tableau à un élément dans deux cas selon la valeur choisie puis la recherche de chacune des bornes du tableau de l'exemple.

5. Pour mettre en œuvre les connaissances de la séquence précédente.

Bases de données, langage SQL

EXERCICE 1 ☞

1. Déterminez une modélisation relationnelle d'un bulletin scolaire qui doit permettre de mentionner
   - des élèves
   - un ensemble d'enseignements
   - au plus une note et une appréciation par enseignement et par élève.
2. Identifiez précisément les contraintes d'intégrité de ce modèle et leurs effets dans le contexte.

EXERCICE 2 ☞

On cherche à modéliser des informations sur les départements français.

1. Concevez une modélisation relationnelle après avoir listé une série d'attributs adaptés à laquelle vous ajouterez la liste de tous les départements voisins.
2. Proposez une contrainte utilisateur qui permette d'éviter le redondance d'information dans la liste des voisins.

EXERCICE 3 ☞

On cherche à modéliser un réseau de bus.

Proposez une modélisation relationnelle assez riche afin de permettre de générer, pour chaque arrêt de bus du réseau, une fiche horaire avec tous les horaires de passage de toutes les lignes de bus qui desservent l'arrêt.

EXERCICE 4 ☞

On considère deux relations

$$R(\underline{a}\ \texttt{Int},\ b\ \texttt{Int},\ c\ \texttt{Int})$$
$$S(\underline{a}^{*}\ \texttt{Int},\ \underline{e}\ \texttt{Int})$$

Dîtes si les affirmations suivantes sont vraies ou fausses en justifiant votre réponse.

1. Les a de R sont deux à deux distincts.
2. Les b de R sont deux à deux distincts.
3. Les a de S sont deux à deux distincts.
4. Les e de S sont deux à deux distincts.
5. S peut être vide alors que R est non vide.
6. R peut être vide alors que S est non vide.

EXERCICE 5 ☞

1. Donnez les ordres SQL permettant de créer les tables de l'exercice n° 1.
2. Donnez les ordres SQL permettant de créer les tables de l'exercice n° 2.
   Donnez des ordres SQL permettant d'ajouter des données valides dans ces tables.

EXERCICE 6 ☞

On considère les deux tables suivantes :

```
CREATE TABLE joueur (
  id_j INT PRIMARY KEY,
  nom VARCHAR(100) NOT NULL );

CREATE TABLE partie (
  j1 INT REFERENCES joueur(id_j),
  j2 INT REFERENCES joueur(id_j),
  score1 INT NOT NULL,
  score2 INT NOT NULL,
  PRIMARY KEY(j1, j2),
  CHECK (j1 <> j2) );
```

qui stockent les résultats de parties entre deux joueurs.

1. Listez toutes les contraintes d'intégrité et proposez pour chacune d'entre elles un ordre SQL qui la viole.
2. Modifiez ces ordres de création afin que
   - la table *partie* contienne en plus un attribut *jour* non nul indiquant la date à laquelle la partie a lieu ;
   - les scores ne puissent être négatifs ;
   - deux joueurs ne puissent pas s'affronter plusieurs fois le même jour.

Les exercices qui suivent reposent sur la base de données `mediatheque.db` disponible dans le cours en ligne.

Au lycée et sur un ordinateur personnel après installation du logiciel, nous utiliserons *DB Browser for SQLite* téléchargeable depuis cette page WEB https://sqlitebrowser.org/.

Les ordres SQL peuvent aussi être testés avec l'application en ligne

https://sqliteonline.com/

en commençant par télécharger et ouvrir la base de données bien sûr.

EXERCICE 7 ☞

Des requêtes simples, sans jointure ni imbrication.
Écrivez le code `sql` des requêtes qui permettent d'obtenir les résultats demandés.

1. Tous les titres des livres de la médiathèque.
2. Tous les noms des usagers de la médiathèque.
3. Tous les noms des usagers de la médiathèque en retirant les doublons.
4. Les titres de tous les livres publiés avant 1980.
5. Les codes isbn des livres à rendre pour le 28/12/2023.

6. Les noms de tous les auteurs recensés par la médiathèque triés par ordre alphabétique.
7. Les noms des usagers habitant dans les 12e et 13e arrondissements de Paris.
   Les codes postaux sont respectivement 75012 et 75013.
8. Les noms et adresses des usagers n'habitant pas une rue.

   Autrement dit, la chaîne de caractères « rue » ne doit pas figurer dans le libellé de l'adresse.
9. Les années et les titres des livres parus durant une année bissextile.

   Rappel : le rang d'une année bissextile est un nombre entier divisible par 4 mais pas par 100 sauf s'il est divisible par 400.

EXERCICE 8 ☞

Des requêtes plus complexes, avec jointure ou imbrication.
Écrivez le code `sql` des requêtes qui permettent d'obtenir les résultats demandés.

1. Les titres de tous les livres empruntés.
2. Les titres des livres empruntés qui sont à rendre avant le 31/12/2023.
3. Le nom et le prénom de l'auteur du livre « 1984 ».
4. Sans doublon, le nom et le prénom des usagers ayant emprunté des livres.
5. Reprise de la question précédente en triant les noms par ordre alphabétique.
6. Les titres de tous les livres publiés strictement avant « Dune ».
7. Les noms et les prénoms des auteurs des livres trouvés par la requête précédente.
8. Reprise de la question précédente en supprimant les doublons.
9. Le nombre de résultats trouvés à la question précédente.

EXERCICE 9 ☞

Formulez en français ce que réalisent les requêtes SQL qui suivent.

1. Une sélection.

```sql
SELECT * FROM livre WHERE titre LIKE '%Robot%';
```

2. Une sélection et une projection.

```sql
SELECT nom, prenom FROM usager WHERE ville = 'Guingamp';
```

3. Une sélection et une projection avec une jointure.

```sql
SELECT u.nom, u.prenom
  FROM usager AS u
  JOIN emprunt AS e ON u.code_barre = e.code_barre
  WHERE retour < '2023-11-22'
;
```

4. Une sélection et une projection avec une requête imbriquée.

```sql
SELECT l.titre
  FROM livre AS l
  WHERE l.isbn IN
    (SELECT isbn FROM livre WHERE annee > 1990)
;
```

EXERCICE 10 ☞

Des requêtes de mise à jour pour terminer.
Écrivez le code `sql` des requêtes qui permettent de réaliser les tâches demandées.

1. Ajoutez un nouvel usager, il s'agit de Soulayi Fatima habitant au n°8 du boulevard Magenta dans le 10e arrondissement de Paris.
2. Il y avait erreur sur le prénom, c'est en fait Maryam.
   Écrivez une requête qui réalise la correction.
3. Ajoutez deux années et huit mois à toutes les dates de retour de la base.
   Vous pourriez utiliser la fonction `DATE_ADD` comme dans cet exemple

   ```sql
   SELECT DATE_ADD(<attribut_date>, INTERVAL 2 YEAR) FROM <nom_table>;
   ```

   mais elle n'est pas implémentée avec SQLITE.
   Vous trouverez ici un équivalent.
4. Mme Soulayi souhaite emprunter l'ouvrage intitulé « La Nuit des temps » de René Barjavel.
   Écrivez une requête qui réalise cet emprunt avec une date de retour fixée au 25/12/2023.
5. La médiathèque étant exceptionnellement fermée du 25/12/2023 compris au 02/01/2024 non compris, les prêts s'achevant dans cette période sont prolongés jusqu'au 05/01/2024.
   Écrivez une requête qui réalise cette prolongation.
6. Vérifiez qu'il existe deux enregistrements pour l'auteur René Barjavel.
   Écrivez une requête pour supprimer l'enregistrement en trop.

CORRIGÉ EXERCICE 1

Modélisation d'un bulletin scolaire

1. La modélisation relationnelle peut consister en cinq relations :

Eleve(*nom* `String`, *prénom* `String`, *date_naissance* `Date`, *el_id* `Int`)
Periode(*intitulé* `String`, *pe_id* `Int`)
Enseignement(*intitulé* `String`, *en_id* `Int`)
Note(*el_id** `String`, *en_id** `Int`, *pe_id** `Int`, *valeur* `Float`)
Appreciation(*el_id** `String`, *en_id** `Int`, *pe_id** `Int`, *libellé* `String`)

On pourrait aussi ajouter une relation *Classe*, qui correspond à une entité, et une autre *EnClasse*, qui correspond à une association. De même pour une relation *Enseignant*, comme entité, et une autre *Enseigne* comme association.

2. En ne considérant que les cinq relations détaillées à la question précédente, voici ce que l'on peut dire des contraintes d'intégrité.

- Pour *Eleve*, la clé primaire assure l'unicité de chaque enregistrement donc cette relation est compatible avec des homonymes né.es le même jour.
- Pour *Periode*, on a choisi d'imposer une clef primaire pour laisser libre l'expression de l'intitulé. L'intitulé aurait pu constituer une clef primaire mais avec un risque d'erreur lié à des saisies à l'usage.
- Pour *Enseignement*, même constat.
- Pour *Note* et *Appreciation*, il s'agit d'associations entre les trois relations précédentes, qui correspondent à des entités, avec ajout d'un attribut propre.
  La clef primaire est donc constituer de l'union des clefs étrangères des entités associées conformément aux règles énoncées dans le cours.

Avec ces contraintes d'intégrités, il est possible de distinguer les élèves, les périodes et les enseignements pour attribuer une note et une appréciation de manière indépendante à chaque individualité triplet.

CORRIGÉ EXERCICE 2

Modéliser les départements français

1. On peut penser à stocker dans une entité le nom du département, son code, son chef-lieu, sa préfecture, son préfet et donc la liste des départements voisins qui relèvera d'une association.
Attention, le code est une chaîne de caractères à cause des deux départements corses et de ceux commençant par un chiffre 0.
Pour les préfets, une entité dédiée leur est consacrée afin de permettre la prise en compte des changements d'affectation des préfets. On distingue alors à nouveau une entité, le préfet, d'une association, l'affectation d'un préfet dans un département.
Une entité pour les régions est aussi possible avec une association pour modéliser l'appartenance d'un département à une région. Pour cette entité, on retiendra le nom de la région et son code.
La nomenclature de l'INSEE est consultable sur cette page WEB.

Departement(*nom* `String`, *code* `String`, *chef_lieu* `String`, *préfecture* `String`)
Prefet(*nom* `String`, *prénom* `String`, *position* `String`, *pre_id* `Int`)
Region(*nom* `String`, *code* `String`)
AffectePrefet(*code** `String`, *pre_id** `Int`, *date* `Date`)
DansRegion(*reg** `String`, *dep** `String`)
Voisin(*dep1** `String`, *dep2** `String`)

où *dep1* désigne le *code* du premier département et *dep2* celui du second.

Cette relation *Voisin* correspond à une association entre deux éléments de la relation *Departement*, ainsi la clef primaire est-elle l'union des deux clefs étrangères identifiant chacun des deux départements voisins.

2. Pour éviter la redondance d'information dans la relation *Voisins*, on va imposer la contrainte utilisateur : « *dep1* est plus petit que *dep2* ».

   Les chaînes de caractères sont comparables selon l'ordre lexicographique.

CORRIGÉ EXERCICE 3

Modéliser un réseau de bus

On commence par déterminer les informations pertinentes.

- Un arrêt de bus peut être représenté par ses coordonnées GPS, soit deux nombres décimaux, son nom et un identifiant unique.
- Une ligne de bus peut être représentée par son nom et un identifiant unique.
- Pour chaque ligne et chaque arrêt desservi par cette ligne on stocke les horaires ainsi que le jour de validité de cet horaire.

On parvient ainsi à la modélisation suivante avec trois relations.

Arret(*lat* `Float`, *lon* `Float`, *nom* `String`, *ar_id* `Int`)
Ligne(*num* `Int`, *intitulé* `String`)
Horaire(*num** `Int`, *ar_id** `Int`, *heure* `Time`, *jour* `Date`)

La relation *Horaire* est une association entre les deux entités *Arret* et *Ligne* complétée par deux attributs qui lui sont propres.

Les clefs primaires des deux entités associées, clefs étrangères de l'association, constituent donc avec leur union la clef primaire de cette association selon les règles du cours.

Cependant, pour un même arrêt et un même numéro de ligne, cette relation *Horaire* doit pouvoir enregistrer plusieurs horaires et jours. Cette union ne peut donc suffire pour réaliser la clef primaire. Un horaire de passage est en fait identifié de manière unique par tous ses attributs et ils constituent donc ensemble la clef primaire de la relation *Horaire*.

CORRIGÉ EXERCICE 4

Vrai - Faux pour réfléchir un peu

1. Vrai car *a* est une clef primaire.
2. Faux car *b* n'intervient pas dans la clef primaire de la relation *R*.
3. Faux car, seul l'attribut *a* est une clef étrangère. La seule contrainte est donc qu'il doit avoir la valeur d'un *a* de la relation *R*.
4. Faux car, seul l'attribut *e* n'a pas de contrainte. Comme il participe à la clef primaire de la relation *S*, ce sont donc les couples (a, e) qui doivent être deux à deux distincts.
5. Vrai, si la relation *S* est vide alors elle respecte bien la propriété « chaque valeur de *a* de *S* référence une valeur de *a* de *R* ».
6. Faux car si la relation *S* est non vide alors elle contient au moins une entité et l'attribut *a* de cette entité doit référencer un *a* existant dans la relation *R* qui ne peut donc pas être vide.

CORRIGÉ EXERCICE 5

Création et remplissage de tables

1. Voici deux exemples d'ordres SQL pour créer les tables représentant les relations de l'exercice n° 1.

```
CREATE TABLE eleve(
  nom TEXT,
  prenom TEXT,
  date_naissance DATE,
  id_el INT PRIMARY KEY );
```

Pour la création de l'une des trois tables implémentant une entité.

```
CREATE TABLE note(
  id_el INT REFERENCES eleve(id_el),
  id_pe INT REFERENCES periode(id_pe),
  id_en INT REFERENCES enseignement(id_en),
  valeur DECIMAL(3,1),
  PRIMARY KEY(id_el, id_pe, id_en) );
```

Pour la création de l'une des deux tables implémentant une association.

2. Voici deux exemples d'ordre SQL pour créer les tables représentant les relations de l'exercice n° 2.

```
CREATE TABLE departement(
  nom TEXT,
  code TEXT PRIMARY KEY,
  chef_lieu TEXT,
  prefecture TEXT );
```

Pour la création d'une table implémentant une entité.

```
CREATE TABLE voisin(
  dep1 TEXT REFERENCES departement(code),
  dep2 TEXT REFERENCES departement(code),
  PRIMARY KEY(dep1, dep2),
  CHECK (dep1 < dep2) );
```

Pour la création de la table implémentant la relation *Voisin* avec l'ajout d'une contrainte métier.
Voici deux exemples d'ordres pour insérer des données valides dans deux tables de l'exercice n° 2.

```
INSERT INTO departement VALUES
  ('Ain', '01', 'Bourg-en-Bresse', 'Bourg-en-Bresse'),
  ('Haute-Savoie', '74', 'Annecy', 'Annecy'),
  ('Savoie', '73', 'Chambéry', 'Chambéry'),
  ('Isère', '38', 'Grenoble', 'Grenoble'),
  ('Rhône', '69', 'Lyon', 'Lyon'),
  ('Saône et Loire', '71', 'Mâcon', 'Mâcon'),
  ('Jura', '39', 'Lons-Le-Saunier', 'Lons-Le-Saunier');

INSERT INTO voisin VALUES
  ('01', '39'), ('01', '71'), ('01', '69'), ('01', '38'), ('01', '73'),
  ('01', '74'), ('39', '71'), ('69', '71'), ('38', '69'), ('38', '73'),
  ('73', '74');
```

CORRIGÉ EXERCICE 6

Les joueurs

1. On liste les contraintes d'intégrité en proposant à chaque fois un ordre SQL contraire.
   - *id_j* est une clef primaire, contrainte d'entité
     Deux ordres, le premier insère des données valides, le second viole la contrainte d'entité car l'identifiant doit être unique.

     ```
     INSERT INTO joueur VALUES (1, 'Benzema'), (2, 'Griezman');
     INSERT INTO joueur VALUES (1, 'Giroud');
     ```

   - *nom* doit être non nul, contrainte de domaine
     Deux ordres, le premier insère des données valides, le second viole la contrainte de domaine car la valeur du champ *nom* ne peut pas être `NULL`.

     ```
     INSERT INTO joueur VALUES (1, 'Benzema'), (2, 'Griezman');
     INSERT INTO joueur VALUES (3, NULL);
     ```

   - *j1* et *j2* sont des clefs étrangères, contrainte de référence
     Trois ordres, les deux premiers insèrent des données valides, le troisième viole la contrainte de référence car la valeur du champ *j1* ou *j2* doit figurer dans la table *joueur*.

     ```
     INSERT INTO joueur VALUES
       (1, 'Benzema'), (2, 'Griezman'), (3, 'Mbappe');
     INSERT INTO partie VALUES (1, 2, 5, 4), (1, 3, 6, 6);
     INSERT INTO partie VALUES (1, 4, 9, 9);
     ```

   - *j1* et *j2* doivent avoir des valeurs différentes, contrainte utilisateur.
     Trois ordres, les deux premiers insèrent des données valides, le troisième viole la contrainte utilisateur car les valeurs des champs *j1* et *j2* doivent être distinctes tout en figurant dans la table *joueur*.

     ```
     INSERT INTO joueur VALUES
       (1, 'Benzema'), (2, 'Griezman'), (3, 'Mbappe');
     INSERT INTO partie VALUES (1, 2, 5, 4), (1, 3, 6, 6);
     INSERT INTO partie VALUES (1, 1, 9, 9);
     ```

2. On modifie la création de la table *partie* en ajoutant un champ *jour*.

```
CREATE TABLE partie (
  j1 INT REFERENCES joueur(id_j),
  j2 INT REFERENCES joueur(id_j),
  score1 INT NOT NULL,
  score2 INT NOT NULL,
  jour DATE NOT NULL,
  UNIQUE (j1, j2, jour),
  PRIMARY KEY(j1, j2),
  CHECK ( (j1 < j2) AND (score1 >= 0) AND (score2 >= 0) ) );
```

L'utilisation de `UNIQUE` va assurer l'unicité du triplet dans la table sans modifier la clef primaire.

On force aussi l'identifiant du joueur 1 à être inférieur à celui du joueur 2 pour éviter une partie j1, j2 et une partie j2, j1 comme on l'avait fait pour les départements voisins.

CORRIGÉ EXERCICE 7

Requêtes simples

Des requêtes simples, sans jointure ni imbrication.

1. Tous les titres des livres de la médiathèque.

```
SELECT titre FROM livre;
```

2. Tous les noms des usagers de la médiathèque.

```
SELECT nom FROM usager;
```

3. Tous les noms des usagers de la médiathèque en retirant les doublons.

```
SELECT DISTINCT nom FROM usager;
```

4. Les titres de tous les livres publiés avant 1980.

```
SELECT titre FROM livre WHERE annee < 1980;
```

5. Les codes isbn des livres à rendre pour le 28/12/2023.

```
SELECT isbn FROM emprunt WHERE retour = "2023-12-28";
```

6. Les noms de tous les auteurs recensés par la médiathèque triés par ordre alphabétique.

```
SELECT nom FROM auteur ORDER BY nom;
```

   On rappelle que l'ordre est croissant par défaut, ce qui correspond à l'ordre alphabétique pour des chaînes de caractères.

7. Les noms des usagers habitant dans les 12e et 13e arrondissements de Paris.

```
SELECT nom FROM usager WHERE cp = '75012' OR cp = '75013';
```

8. Les noms et adresses des usagers n'habitant pas une rue.

```
SELECT nom, adresse FROM usager WHERE adresse NOT LIKE '%rue%';
```

9. Les années et les titres des livres parus durant une année bissextile.

```
SELECT annee, titre FROM livre
  WHERE ( ( (annee%4 = 0) AND (annee%100 <> 0) ) OR (annee%400 = 0) );
```

CORRIGÉ EXERCICE 8

Requêtes plus complexes

Des requêtes plus complexes, avec jointure ou imbrication.

1. Les titres de tous les livres empruntés.

```
SELECT l.titre FROM livre AS l
  JOIN emprunt AS e ON l.isbn = e.isbn;
```

   Il est nécessaire de réaliser une jointure entre les deux tables *livre* et *emprunt*. Cette jointure produit tous les livres empruntés avec tous les champs des deux tables.
   On réalise une projection en ne conservant que les valeurs du champ *titre*.

2. Les titres des livres empruntés qui sont à rendre avant le 31/10/2023.

```
SELECT l.titre FROM livre AS l
  JOIN emprunt AS e ON l.isbn = e.isbn
  WHERE e.retour < "2023-10-31" ;
```

   La même requête en ajoutant une condition avec la clause `WHERE`.

3. Le nom et le prénom de l'auteur du livre « 1984 ».

```
SELECT a.nom, a.prenom FROM auteur AS a
  JOIN auteur_de AS ad ON a.a_id = ad.a_id
  JOIN livre AS l ON ad.isbn = l.isbn
  WHERE l.titre = "1984" ;
```

   Il est nécessaire de joindre trois tables puis d'imposer une condition.

4. Sans doublon, le nom et le prénom des usagers ayant emprunté des livres.

```
SELECT DISTINCT u.nom, u.prenom FROM usager AS u
  JOIN emprunt AS e ON e.code_barre = u.code_barre;
```

   Même structure que pour les titres de tous les livres empruntés avec une clause `DISTINCT` pour éviter les doublons c'est-à-dire les usagers ayant emprunté plusieurs ouvrages.

5. Reprise de la question précédente en triant les noms par ordre alphabétique.

```
SELECT DISTINCT u.nom, u.prenom FROM usager AS u
  JOIN emprunt AS e ON e.code_barre = u.code_barre
  ORDER BY u.nom;
```

   La clause `ORDER BY` est placée à la fin de la requête, elle est réalisée en dernier sur l'ensemble des enregistrements renvoyés.

6. Les titres de tous les livres publiés strictement avant « Dune ».

```
SELECT titre FROM livre
  WHERE annee < (SELECT annee FROM livre WHERE titre = 'Dune');
```

   On utilise une requête imbriquée comme condition de la clause `WHERE` pour renvoyer l'année de publication du livre `'Dune'`.

7. Les noms et les prénoms des auteurs des livres trouvés par la requête précédente.

```
SELECT a.nom, a.prenom FROM auteur AS a
  JOIN auteur_de AS ad ON a.a_id = ad.a_id
  JOIN livre AS l ON l.isbn = ad.isbn
  WHERE l.annee < (SELECT annee FROM livre WHERE titre = 'Dune')
  ORDER BY a.nom;
```

Il suffit de réaliser les trois jointures avant la clause `WHERE` et d'adapter la projection.
Le tri permet de mieux montrer les doublons.

8. Reprise de la question précédente en supprimant les doublons.

```
SELECT DISTINCT a.nom, a.prenom FROM auteur AS a
  JOIN auteur_de AS ad ON a.a_id = ad.a_id
  JOIN livre AS l ON l.isbn = ad.isbn
  WHERE l.annee < (SELECT annee FROM livre WHERE titre = 'Dune');
```

Avec ajout d'une clause `DISTINCT` pour supprimer les doublons

9. Le nombre de résultats trouvés à la question précédente.

```
SELECT COUNT(*) AS TotalAuteurs FROM
  (SELECT DISTINCT a.nom, a.prenom FROM auteur AS a
    JOIN auteur_de AS ad ON a.a_id = ad.a_id
    JOIN livre AS l ON l.isbn = ad.isbn
    WHERE l.annee < (SELECT annee FROM livre WHERE titre = 'Dune') );
```

On peut considérer la requête de la question précédente comme une requête imbriquée sur le résultat de laquelle on applique la fonction d'agrégation `COUNT`.
L'ajout d'un alias peut rendre le résultat plus présentable.

CORRIGÉ EXERCICE 9

Analyses de syntaxes

1. Une sélection.
Sélectionner tous les enregistrements de la table *livre* dont le titre contient le mot « robot », quelle que soit la casse, et afficher les valeurs de tous les champs.

2. Une sélection et une projection.
Sélectionner tous les enregistrements de la table *usager* dont la valeur du champ *ville* a pour valeur « Guingamp » et afficher uniquement les valeurs des champs *nom* et *prenom*.

3. Une sélection et une projection avec une jointure.
Joindre les deux tables *usager* et *emprunt* sur la clef `code_barre` commune et sélectionner tous ceux dont l'emprunt a une date de retour strictement inférieure au 22/11/2023. N'afficher que les valeurs des champs *nom* et *prenom* pour cette sélection.

4. Une sélection et une projection avec une requête imbriquée.
Sélectionner tous les enregistrements de la table *livre* dont l'isbn figure parmi les valeurs renvoyées par la sélection de tous les isbn des livres dont l'année de parution est strictement postérieure à 1990. N'afficher que la valeur du champ *titre* pour les résultats de cette sélection.
Cette requête peut donc être exprimée plus simplement de la manière suivante :

```
SELECT titre FROM livre
  WHERE annee > 1990 ;
```

CORRIGÉ EXERCICE 10

Mises à jour

1. Il s'agit d'une création, il faut donc insérer une nouvelle valeur à la table *usager*.

```
INSERT INTO usager VALUES
  ('SOULAYI', 'Fatima', '8, Boulevard Magenta', '75010', 'Paris',
   'Fatima.Soulayi@protonmail.com', '123456789101112');
```

2. L'enregistrement existe, il s'agit donc cette fois d'une mise à jour. Attention à ne pas oublier la clause `WHERE` pour n'appliquer le changement qu'à l'usagère considérée.

```
UPDATE usager
  SET prenom = 'Maryam'
  WHERE code_barre = '123456789101112';
```

3. Après une lecture attentive des documentations proposées en liens, on exécute la requête suivante

```
UPDATE emprunt
  SET retour = DATE(retour, '+2 year', '+ 8 month');
```

4. Pour réaliser cet emprunt, il faut récupérer l'isbn du livre, ce qui passe par une requête imbriquée car il n'est pas encore possible de joindre l'usagère au livre sachant que l'emprunt n'a pas encore été réalisé.
   Il s'agit d'un ajout dans la table *emprunt* et non d'une mise à jour.
   On peut considérer que le code barre de l'usagère est connu pour simplifier.

```
  INSERT INTO emprunt VALUES (
    '123456789101112',
    ( SELECT l.isbn FROM livre AS l
      JOIN auteur_de AS ad ON ad.isbn = l.isbn
      JOIN auteur AS a ON a.a_id = ad.a_id
      WHERE a.nom LIKE 'Barjavel' AND a.prenom LIKE 'René' AND l.titre ...
   ...LIKE 'La nuit des temps' ),
    '2023-12-25') ;
```

   Dans une situation réelle, l'usager est identifié par son code barre et le livre aussi.

5. Il s'agit d'une mise à jour de la valeur du champ *retour* dans la table *emprunt* en imposant une condition.

```
UPDATE emprunt
  SET retour = '2024-01-05'
  WHERE retour >= '2023-12-25' AND retour < '2024-01-02' ;
```

6. On identifie le doublon en utilisant l'opérateur `LIKE` et le joker %.

```
SELECT * FROM auteur WHERE nom LIKE 'barjavel%';
```

   Il faut mettre à jour la table *auteur_de* en remplaçant la valeur de *a_id* 13 par 14 pour rapatrier tous les ouvrages au même enregistrement de l'auteur avant de supprimer le doublon par son identifiant.

```
UPDATE auteur_de SET a_id = 14 WHERE a_id = 13 ;
DELETE FROM auteur WHERE a_id = 13 ;
```

Structures de données hiérarchiques

EXERCICE 1 ☞

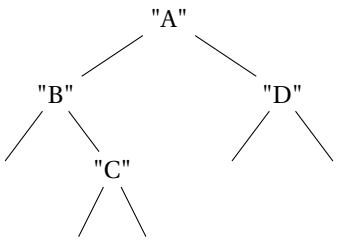
Déterminez le nombre d'arbres binaires contenant exactement 0 nœud, 1 nœud, 2 nœuds, 3 nœuds, 4 nœuds et 5 nœuds.

EXERCICE 2 ☞

Dessinez tous les arbres binaires ayant respectivement 3 et 4 nœuds.

EXERCICE 3 ☞

On se donne un exemple d'arbre binaire pour raisonner.



On souhaite implémenter une structure de données d'arbre binaire à l'aide de tableaux dynamiques emboîtés.

On convient pour cela qu'un arbre binaire vide est modélisé par `[None, [], []]`.

1. Complétez le code de la fonction `creer_ab` puis écrivez une instruction permettant d'instancier l'exemple d'arbre binaire avec cette fonction.
2. Complétez les codes des fonctions nommées `est_vide` et `est_feuille` conformément à leurs spécifications.
3. Complétez le code de la fonction récursive nommée `taille_ab` qui renvoie la taille d'un arbre binaire passé en argument effectif.
4. Complétez le code de la fonction récursive nommée `hauteur_ab` qui renvoie la hauteur d'un arbre binaire passé en argument effectif.
5. Complétez le code de la fonction récursive nommée `afficher_ab` qui renvoie une chaîne de caractères représentant un arbre binaire passé en argument effectif conçue selon la discipline suivante :
   - la chaîne est vide si l'arbre est vide,
   - pour un nœud, on écrit une parenthèse ouvrante puis, récursivement son sous-arbre gauche, sa valeur, récursivement son sous-arbre droit et enfin une parenthèse fermante.

   Pour l'arbre binaire donné en exemple, on attend donc `((B(C))A(D))`.

6. Complétez le code de la fonction nommée `nbFeuilles_ab` qui renvoie le nombre de feuilles d'un arbre binaire passé en argument effectif.

   Pour le parcours en largeur, vous utiliserez le module `queue` dont la documentation officielle est consultable sur cette page WEB.
7. Complétez le code de la fonction récursive nommée `appartient_ab(val, ab)` qui renvoie la valeur `True` si celle de `val` est celle d'un nœud de l'arbre binaire `ab` et `False` sinon.
8. Parmi les codes récursifs précédents, identifiez le type de parcours en profondeur réalisé.

EXERCICE 4 ☞

Dans cet exercice, on va implémenter un arbre binaire de recherche avec un seul tableau dynamique en observant la discipline suivante :

- la valeur de la racine de l'arbre est stockée à l'indice 0 du tableau,
- si l'élément d'indice i contient la valeur d'un nœud alors celui d'indice $2 \times i + 1$ contient celle de l'enfant gauche et celui d'indice $2 \times i + 2$ contient celle de l'enfant droit.

La valeur `None` signifie l'absence de valeur pour la racine ou pour un enfant.

Le fichier du module, nommé `nsi_abrTableau.py`, contient deux fonctions privées dont le code est fourni entièrement écrit :

- `_completer_tab` à exploiter pour l'insertion d'une valeur,
- `_purger_tab` à exploiter pour la suppression d'une valeur.

1. Écrivez le tableau dynamique modélisant un arbre binaire de recherche qui est vide.
2. Écrivez les tableaux dynamiques modélisant plusieurs arbres binaires de recherche construits uniquement avec les quatre nombres entiers 1, 2, 3 et 4 sans doublon.
3. Complétez les codes des deux fonctions nommées `_preconditions` et `creer_abr`.
4. Complétez les codes des fonctions nommées `est_vide` et `est_feuille` conformément à leurs spécifications.
5. Complétez les codes des trois fonctions récursives suivantes :
   a. `taille` qui renvoie la taille d'une instance,
   b. `hauteur` qui renvoie la hauteur d'une instance,
   c. `afficher` qui renvoie une chaîne de caractères formée comme dans l'exercice précédent.
6. Complétez le code de la fonction récursive nommée `inserer` qui insère une nouvelle clef dans un arbre binaire de recherche.
7. Testez soigneusement l'ensemble des fonctions créées jusqu'à présent.
8. Complétez le code de la fonction récursive nommée `appartient` qui renvoie `True` ou `False` selon que la valeur passée en argument soit une clef de l'arbre binaire de recherche passé en second argument ou non.

   Testez soigneusement cette nouvelle fonction.
9. **Pour aller plus loin**.
   a. Complétez le code de la fonction nommée `supprimer` qui va supprimer la première occurrence de la valeur passée en argument si elle est une clef de l'arbre binaire de recherche passé en second argument et ne rien faire sinon.

      Attention à la décomposition en trois sous fonctions récursives et un algorithme principal.

b. Testez très soigneusement cette fonction avec des exemples bien choisis et comportant des doublons.

EXERCICE 5 ☞

En cours et sur un exemple vous avez pu vérifier que le parcours infixe d'un arbre binaire de recherche renvoie les valeurs de ses nœuds rangées en ordre croissant.

L'objectif de cet exercice est d'exploiter cette propriété pour trier un tableau de valeurs comparables.

Le fichier du module se nomme `nsi_abrTri.py`.

1. Écrivez le code de la fonction privée nommée `_est_trie` qui renvoie `True` si le tableau passé en argument est trié.

Cette fonction sera utile pour contrôler des résultats.

2. Écrivez le code de la fonction privée nommée `_parcoursInfixe` qui réalise le parcours en profondeur infixe de l'arbre binaire de recherche passé en argument en renvoyant un tableau constitué des valeurs de ses nœuds ajoutées dans l'ordre du parcours.

Testez cette fonction avec quelques exemples d'arbres binaires de recherche.

3. Complétez le code de la fonction nommée `genere_tab_alea` conformément à sa spécification.

Vous utiliserez la fonction `randint` du module `random`.

4. Écrivez enfin le code de la fonction nommée `tri_tab` qui va renvoyer un nouveau tableau constitué des valeurs des éléments du tableau passé en argument mais triées en ordre croissant.
L'algorithme est simple :

① insertion des valeurs des éléments du tableau dans un arbre binaire de recherche,

② parcours infixe de cet arbre avec renvoi des valeurs de ses nœuds dans l'ordre du parcours.

5. Proposez des tests déterministes puis aléatoires de cette fonction en exploitant les fonctions du module.

6. Quelle est la complexité en temps de cette méthode de tri dans le pire des cas ?

CORRIGÉ EXERCICE 1

Un peu de dénombrement

- Il y a 1 seul arbre binaire avec 0 nœud, c'est l'arbre vide.
- Il y a 1 seul arbre binaire avec 1 nœud, c'est l'arbre réduit à sa racine.
- Il y a 2 arbres binaires avec 2 nœuds, depuis la racine, le deuxième nœud constitue soit le sous-arbre gauche, soit le sous arbre droit.
- Avec 3 nœuds, on place la racine puis les 2 nœuds restants et il n'y a que 3 façons de le faire
  ① les 2 dans le sous-arbre gauche
  ② un dans chacun des deux sous-arbres gauche et droit
  ③ les 2 dans le sous-arbre droit.

  Or on sait qu'il y a 2 arbres binaires possibles avec 2 nœuds, ce qui donne en tout $2+1+2=5$ arbres binaires possibles.
- Avec 4 nœuds, on place la racine puis on détermine les différentes façons de répartir les 3 autres sur les deux sous-arbres gauche et droit : $3+0$, $2+1$, $1+2$, $0+3$.
  En mobilisant les résultats précédents, on trouve $5+2+2+5=14$ arbres binaires possibles.
- En procédant de la même manière pour 5 nœuds, on trouve $4+0$, $3+1$, $2+2$, $1+3$ et $0+4$ répartitions possibles des 4 nœuds distincts de la racine, d'où

$$14+5+2\times 2+5+14=42$$

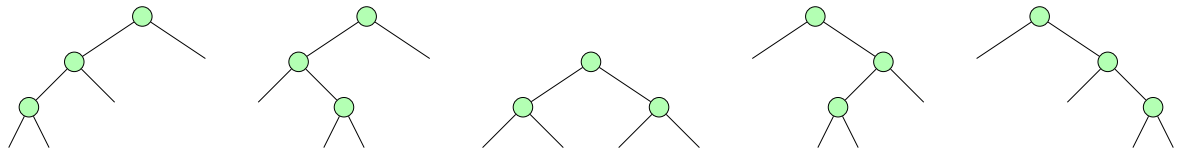
soit 42 arbres binaires possédant 5 nœuds.

CORRIGÉ EXERCICE 2

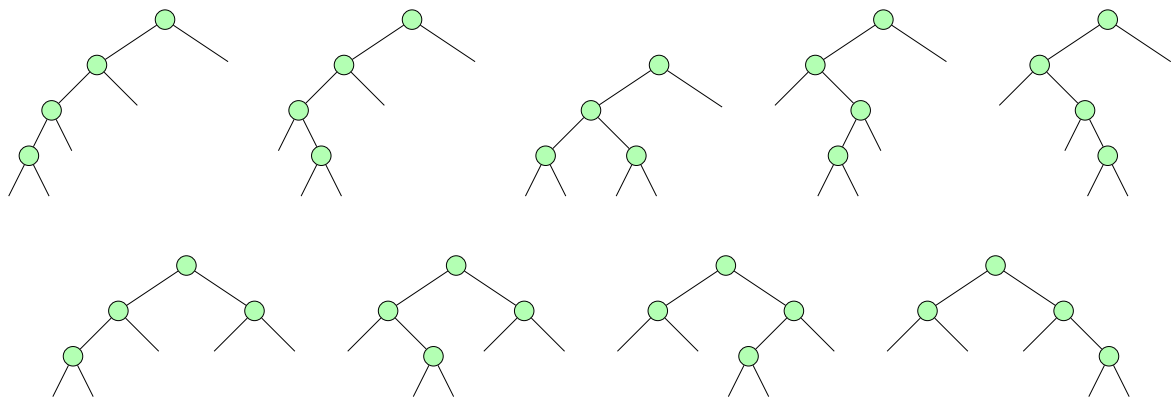
Un peu de représentation

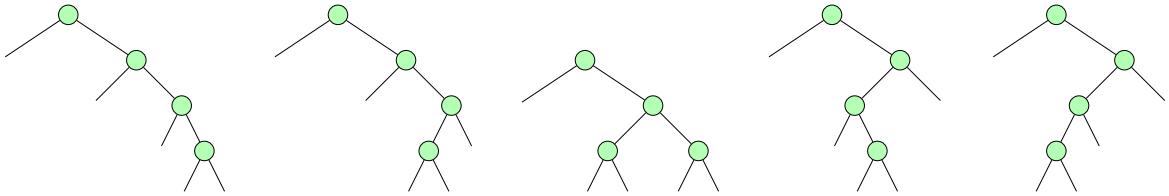
On s'appuie sur les résultats de l'exercice précédent pour ne pas en oublier.

Avec trois nœuds



Avec quatre nœuds





CORRIGÉ EXERCICE 3

Une implémentation d'un arbre binaire

1. Pour cette fonction de création, il suffit de renvoyer un tableau dynamique à trois éléments, les valeurs des variables nommées `r` pour racine, `g` pour sous-arbre gauche et `d` pour sous-arbre droit.
   Les valeurs par défaut permettent d'instancier un arbre vide ou de créer facilement une feuille.
   On compose ensuite les appels dans le bon ordre en exploitant les valeurs par défaut.

   ```
   >>> a = creer_ab('A', creer_ab('B', d = creer_ab('C')), creer_ab('D'))
   ```

   On peut lire cette ligne à partir de la racine pour parcourir séparément chacun des deux sous-arbres gauche et droit récursivement.
2. Il suffit de tester la valeur de chaque élément du tableau dynamique implémentant l'arbre binaire passé en argument.
   Pour les tests d'égalité, on utilise
   - l'opérateur `==` pour savoir si les **valeurs** de deux variables sont égales,
   - l'opérateur `is` pour savoir si les deux variables pointent vers la même adresse en mémoire, donc si elles sont deux alias d'une même constante.

   D'où la distinction entre la comparaison avec `None`, qui est une valeur unique, et celle avec un tableau vide, où il s'agit de deux entités distinctes.
   D'autre part, la comparaison avec `is` est plus rapide.
3. Un algorithme récursif de calcul de la taille d'un arbre binaire a pour **cas de base** la situation d'un arbre binaire vide, et dans ce cas la valeur à renvoyer est 0 ou s'il s'agit d'une feuille, et dans ce cas la valeur à renvoyer est 1.
   Dans le **cas récursif**, l'existence d'une racine conduit à renvoyer la valeur 1 additionnée des tailles des deux sous-arbres gauche et droit calculées par deux appels récursifs mais il faut être prudent.e car, avec l'implémentation choisie, si l'un des sous-arbres est vide sa modélisation est un tableau vide et non un tableau à trois éléments. C'est pourquoi on distingue trois cas d'appels récursifs.
4. La hauteur d'un arbre binaire est égale au maximum des valeurs des profondeurs.
   L'algorithme récursif est analogue au précédent mais en ajoutant 1 au maximum des deux valeurs renvoyées par appels récursifs pour les deux sous-arbres gauche et droit quand aucun des deux n'est vide sinon c'est uniquement la hauteur du sous-arbre non vide.
   Ce maximum est calculable par appel à la fonction native `max` ou par écriture et appel d'une fonction privée du module.
   Le cas de base est identique à celui de la fonction `taille` mais il peut cette fois être regroupé en un seul cas car un arbre réduit à sa seule racine a pour hauteur 0.
5. Cette fonction renvoie une chaîne de caractères.
   L'algorithme récursif est le même que celui du calcul de la taille en adaptant les traitements pour construire une chaîne de caractères telle que demandée.

6. On a choisi de traiter les deux cas de base, arbre vide ou arbre réduit à une racine, avant d'engager le parcours en largeur demandé pour calculer le nombre de feuilles.

Dans ce parcours en largeur, on ne va enfiler que les sous-arbres non vides pour maintenir le traitement d'une structure d'arbre binaire.

Ce calcul peut aussi être réalisé par l'algorithme récursif d'un parcours en profondeur.