CAPES NSI L3 – session 2026

Épreuve disciplinaire

Exercice 1 : ADN, ARN et Acides Aminés

Partie A : Représentation et manipulation de l'ADN

L'ADN est une molécule formée d'une chaîne de nucléotides, chacun contenant l'une des quatre bases azotées suivantes :

- l'adénine, notée `A` ;
- la thymine, notée `T` ;
- la cytosine, notée `C` ;
- la guanine, notée `G`.

Ces petites molécules, appelées nucléotides, servent de *lettres* pour écrire le code génétique. Trois bases consécutives forment ce qu'on appelle un **codon**.
Par exemple, `ATC` est un codon.
On considère le brin d'ADN représenté par la chaîne de caractères `adn` suivante :

```
adn = "ATCGTACGT"
```

1. Quels sont le type et la valeur renvoyés par l'instruction suivante :

   ```
   >>> adn
   ```

2. Quels sont le type et la valeur renvoyée par l'instruction suivante :

   ```
   >>> len(adn)
   ```

3. Quelle est la valeur renvoyée par l'instruction suivante :

   ```
   >>> adn[1:6]
   ```

4. Écrire une fonction `occurrences(adn)` qui prend en paramètre une chaîne de caractères `adn` représentant un brin d'ADN et qui renvoie un dictionnaire associant à chaque caractère représentant un nucléotide son nombre d'occurrences dans le brin d'ADN.
   Par exemple :

   ```
   >>> occurrences(adn)
   {"A" : 2, "T" : 3, "C" : 2, "G" : 2}
   ```

5. Écrire une fonction `appartient(codon, adn)` qui prend en paramètres deux chaînes de caractères `codon` représentant un codon et `adn` représentnant un brin d'ADN et qui renvoie `True` si le codon se trouve dans la brin d'ADN et `False` sinon.
   Par exemple :

   ```
   >>> appartient("TCG", "ATCGTACGT")
   True
   >>> appartient("TCA", "ATCGTACGT")
   False
   ```

6. Écrire une fonction `codons(adn)` qui prend en paramètre une chaîne de caractères `adn` représentant un brin d'ADN et qui renvoie un tableau contenant tous les codons possibles de ce brin.
   Par exemple :

   ```
   >>> codons("ATCGTACGT")
   ["ATC", "TCG", "CGT", "GTA", "TAC", "ACG", "CGT"]
   ```

Partie B : Génération et analyse de séquence

Rappel : La fonction `choice` de la bibliothèque `random` renvoie un élément unique choisi au hasard dans une séquence (liste, tuple, chaîne de caractères, etc...) non vide.
Par exemple :

```
>>> from random import *
>>> choice(["ada", "bob", "eve", "anna"])
"eve"
```

7. Compléter la fonction `generate(n)` qui prend en paramètre un entier `n` et renvoie une chaîne de caractères composée de `n` codons, composés de nucléotides choisis aléatoirement parmi les valeurs `A`, `T`, `G` et `C`.

```
1  import random
2
3  nucleotides = ['A', 'T', 'G', 'C']
4
5  def generate(n):
6      resultat = ""
7      for _ in range(...):
8          codon = ""
9          for _ in range(...):
10             codon = ...
11         resultat = ...
12     return resultat
```

On dispose de la fonction `mystere` suivante :

```
def mystere(adn):
    n = len(adn)
    a, c, g, t = (0, 0, 0, 0)
    i = n-1
    while i >= 0:
        if adn[i] == 'A':
            a += 1
        elif adn[i] == 'C':
            c += 1
        elif adn[i] == 'G':
            g += 1
        elif adn[i] == 'T':
            t += 1
        i -= 1
    return [100*a/n, 100*c/n, 100*g/n, 100*t/n]
```

8. Expliquer brièvement pourquoi cette fonction `mystere` se termine.
9. Déterminer, en détaillant les étapes, ce que renvoie l'appel :

```
>>> mystere("AAATGGC")
```

10. Donner le rôle de cette fonction `mystere`.

Partie C : Transcription et traduction

La transcription est un processus biologique qui consiste à copier une séquence d'ADN en une séquence d'ARN.
Lors de cette étape, l'ADN est transformé en ARN en remplaçant la thymine (`T`) par l'uracile (`U`).

11. Écrire une fonction `transcript(adn)` qui prend en paramètre une chaîne de caractères `adn` représentant un brin d'ADN et qui renvoie une nouvelle chaîne de caractères représentant le brin d'ARN correspondant en remplaçant la thymine (T) par de l'uracile (U).
12. Déterminer ce que renvoie l'appel :

```
>>> transcript("ATGCTA")
```

Chaque codon de l'ARN permet la fabrication d'un acide aminé précis. Le tableau ci-dessous permet d'associer les codons aux acides aminés.

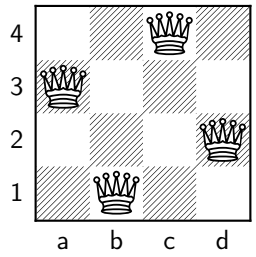
nucléotide N°1	nucléotide N°2									nucléotide N°3
	U		C		A		G			
U	UUU	F	UCU	S	UAU	Y	UGU	C	U	
	UUC		UCC		UAC		UGC		C	
	UUA	L	UCA		UAA	*	UGA	*	A	
	UUG		UCG		UAG		UGG	W	G	
C	CUU	L	CCU	P	CAU	H	CGU	R	U	
	CUC		CCC		CAC		CGC		C	
	CUA		CCA		CAA	Q	CGA		A	
	CUG		CCG		CAG		CGG		G	
A	AUU	I	ACU	T	AAU	N	AGU	S	U	
	AUC		ACC		AAC		AGC		C	
	AUA		ACA		AAA	K	AGA	R	A	
	AUG	M	ACG		AAG		AGG		G	
G	GUU	V	GCU	A	GAU	D	GGU	G	U	
	GUC		GCC		GAC		GGC		C	
	GUA		GCA		GAA	E	GGA		A	
	GUG		GCG		GAG		GGG		G	

Codes		
G	Glycine	Gly
P	Proline	Pro
A	Alanine	Ala
V	Valine	Val
L	Leucine	Leu
I	Isoleucine	Ile
M	Methionine	Met
C	Cysteine	Cys
F	Phenylalanine	Phe
Y	Tyrosine	Tyr
W	Tryptophan	Trp
H	Histidine	His
K	Lysine	Lys
R	Arginine	Arg
Q	Glutamine	Gln
N	Asparagine	Asn
E	Glutamic Acid	Glu
D	Aspartic Acid	Asp
S	Serine	Ser
T	Threonine	Thr

13. Écrire un dictionnaire Python `acide_amin` qui associe à chaque acide aminé la liste de ses codons. On se contentera de donner le début du dictionnaire pour les cinq premiers acide aminés.
14. Déterminer ce que renvoie `acide_amin['leu']`.
15. Écrire une fonction Python `arn_vers_acid_amin` qui prend en paramètre une chaîne de caractères `arn` représentant un brin d'ARN et qui renvoie une liste des acides aminés correspondants en s'arrêtant si un codon `"*"` est rencontré.
16. Déterminer ce que renvoie l'appel `arn_vers_acid_amin("AUGUUUUGA")`.
17. Écrire une fonction Python `spectre(adn)` qui prend en paramètre une chaîne de caractères `adn` représentant un brin d'adn et qui renvoie la liste des codons distincts de longueur 3 contenus dans une séquence d'ADN.
18. Déterminer le spectre de `"ATACTACTA"` ?
19. Écrire une fonction `ordonne_spectre` qui prend en paramètre une liste `spectre` représentant le spectre d'un brin d'adn et qui trie sur place ce tableau suivant l'ordre lexicographique.
20. Donner la complexité dans le pire des cas de votre fonction `ordonne_spectre`.

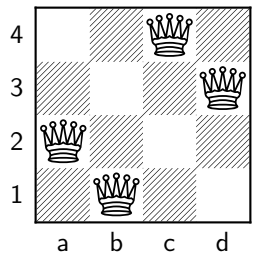
Exercice 2 : Le problème des n reines

Le **problème des `n` reines** consiste à placer `n` reines sur un échiquier `n×n` de telle sorte qu'aucune reine ne puisse en prendre une autre, c'est-à-dire qu'aucune ne soit sur la même ligne, colonne ou diagonale qu'une autre.
Voici, par exemple, une solution du problème des 4 reines :



21. Proposer deux structures de données pour représenter les positions des reines sur l'échiquier.
22. Donner des avantages et des inconvénients pour chacune des deux structures de données choisies.

Pour le problème des 4 reines traité ci-dessous, on utilisera un tableau où chaque case représente une colonne de l'échiquier, et la valeur contenue dans cette case indique la ligne sur laquelle la reine est placée dans cette colonne.
Par exemple, l'échiquier



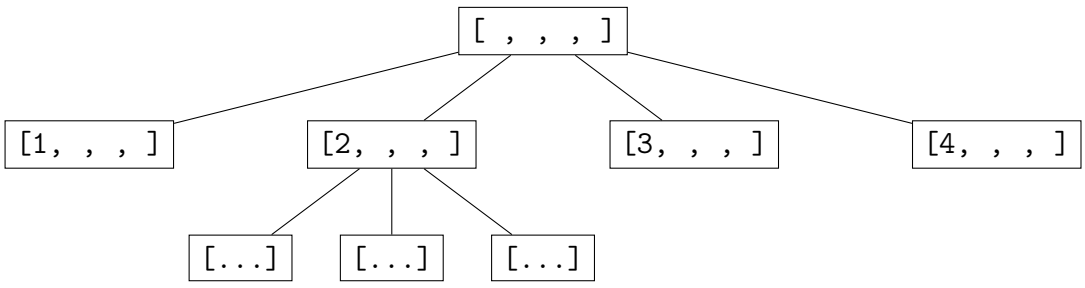
sera représenté par le tableau `[2, 1, 4, 3]`.

23. Déterminer, en justifiant, le tableau représentant la solution présentée en exemple, avec la représentation choisie.

La résolution du problème des 4 reines peut être représenté par un arbre de décision, où

- Chaque nœud représente une configuration partielle ou complète, c'est-à-dire un état de l'échiquier où certaines reines sont déjà placées ;
- Chaque branche correspond à une décision ou tentative d'ajouter une reine dans une colonne spécifique sur la prochaine ligne disponible, en respectant la règle de non-mencae.

24. Reproduire et compléter l'arbre de décision pour obtenir toutes les solutions du problème des 4 reines.



25. Écrire une fonction `est_sans_conflit(pos, new_col, new_lig)` qui prend en paramètres :
    - un tableau `pos` où l'indice représente une colonne de l'échiquier et la valeur stockée à cet indice correspond à la ligne où la reine a été placée dans cette colonne ;
    - deux entiers `new_col` et `new_lig`.

    et qui vérifie si une reine peut être placée dans la colonne `new_col` à la ligne `new_lig` sans être attaquée par les reines déjà placées sur les précédentes colonnes.
    Pour chaque colonne déjà occupée (de `0` à `new_col-1`), la fonction doit vérifier qu'aucune reine n'est sur la même ligne ou sur la même diagonale que la position envisagée.
    Si la position est sûre, la fonction renvoie alors `True` et `False` sinon.

On considère la fonction `mystere` suivante :

```
1 def mystere(n, col, pos, sol):
2     if col == n:
3         sol.append(pos.copy())
4     for lig in range(n):
5         if est_sans_conflit(pos, col, lig):
6             pos[col] = lig
7             mystere(n, col+1, pos, sol)
```

On suppose que les variables `pos` et `sol` ont été initialisées de la façon suivante :

```
>>> pos = [-1 for _ in range(4)]
>>> sol = []
```

26. Que contient la variable `sol` après l'appel suivant :

```
>>> mystere(4, 0, [-1] * 4, [])
```

27. Expliquer le rôle de la fonction `mystere`.
28. Quel type de programmation utilise la fonction `mystere` ? Justifier.
29. Écrire une fonction `est_valide(pos)` qui prend en paramètre un tableau `pos` et qui renvoie `True` si aucune paire de reines n'est sur la même diagonale et `False` sinon.
30. Écrire une fonction `inserer(val, ind, tab)` qui prend en paramètre une valeur `val`, un indice `ind` (supposé compris entre `0` et `len(tab)` et un tableau `tab` et qui renvoie un nouveau tableau dans lequel la valeur `val` a été insérée à l'indice `ind` dans le tableau `tab`.
    Par exemple :

```
>>> inserer(3, 2, [5, 2, 7])
[5, 2, 3, 7]
>>> inserer(6, 4, [5, 2, 3, 7])
[5, 2, 3, 7, 6]
```

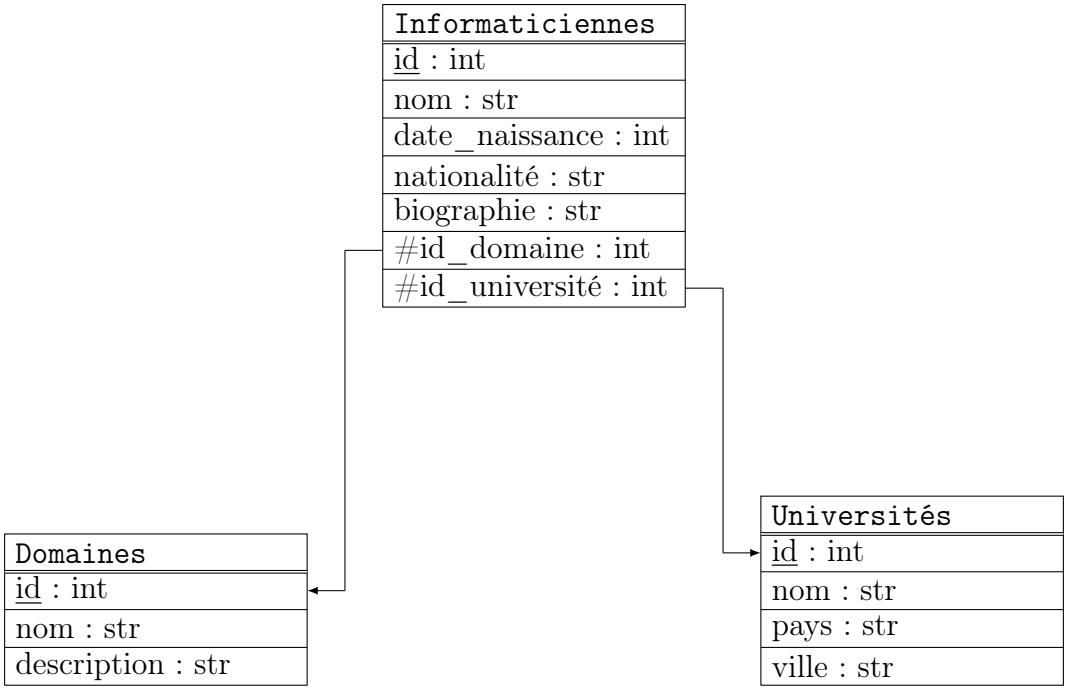
31. Écrire une fonction récursive `permutations(n)` qui prend en paramètre un entier `n` et qui renvoie un tableau contenant tous les tableaux représentant les permutations des entiers de `1` à `n`.
    Par exemple :

```
>>> permutations(2)
[[1, 2], [2, 1]]
>>> permutations(3)
[[1, 2, 3],[1, 3, 2],[2, 1, 3],[2, 3, 1],[3, 1, 2],[3, 2, 1]]
```

    On utilisera la fonction `inserer` de la question précédente.
32. Écrire une fonction `force_brute_reines(n)` qui prend en paramètre un entier `n` et qui renvoie un tableau de tableaux contenant toutes les solutions du problème des `n` reines en testant **toutes** les possibilités de positionnement des `n` reines.
    On utilisera la fonction `permutations` de la question précédente.
33. Déterminer, en justifiant brièvement, la complexité temporelle de la fonction `force_brute_reines` en fonction de `n`.

Exercice 3 : Bases de données et informaticiennes célèbres

On souhaite modéliser une base de données sur des informaticiennes ayant marqué l'histoire de l'informatique (exemples : Ada Lovelace, Grace Hopper, Frances Allen, Radia Perlman, etc ...).
Cette base de données repose sur le schéma relationnel suivant :



La table `Domaines` contient les différents domaines de recherche des universités.

34. Indiquer pour chaque table la ou les clés primaires et la ou les éventuelles clés étrangères.
35. Écrire une requête SQL permettant d'insérer l'informaticienne Barbara Liskov, née le 7 Novembre 1939, connue pour ses contributions fondamentales aux langages de programmation, à la méthodologie de la programmation et aux systèmes distribués, son domaine de recherche étant la programmation et le logiciel, sachant qu'elle travaille pour le MIT.
36. Écrire une requête SQL permettant d'afficher le nombre d'informaticiennes américaines.
37. Ecrire une requête SQL permettant d'obtenir tous les domaines de recherche pour une université donnée.
38. Écrire une requête SQL permettant de lister les informaticiennes avec leur domaine de recherche et leur université.
39. Écrire une requête SQL permettant d'ajouter un nouveau domaine de recherche, par exemple Intelligence Artificielle, dans la table `Domaines`.
40. Écrire une requête SQL permettant de lister toutes les informaticiennes américaines ayant travaillé sur la compilation et ayant obtenu le Prix Turing.
41. Expliquer comment modifier le schéma relationnel pour permettre aux informaticiennes d'avoir plusieurs nationalités.