BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

**Décembre 2025 – Bac Blanc**

# N.S.I.
## Numérique et Sciences Informatiques

Durée de l'épreuve : **3 heures 30**

*L'usage de la calculatrice n'est pas autorisé*

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.
Ce sujet comporte 11 pages numérotées de 1 à 11

## **Exercice 1 :** *(8 points)*

*Cet exercice porte sur les dictionnaires et leurs algorithmes associés, le traitement de données en table, la sécurisation des communications et la programmation en général.*

Lorsque l'énoncé demande la manipulation de la structure de données abstraites `liste`, on utilisera les `list` en Python avec la méthode `append`.

Un club de judo souhaite développer un système d'informations pour faciliter les traitements administratifs en cours d'année (inscriptions, communication, compétitions…).

Cet exercice comporte 2 parties indépendantes.

### Partie A

On pourra utiliser les mots du langage SQL suivants : `SELECT, FROM, WHERE, JOIN ON, INSERT INTO, VALUES, COUNT, DELETE`.

Le club de Judo souhaite proposer à ses adhérents une location de kimonos. Pour cela, il décide de mettre en place une base de données contenant les relations `adherent`, `kimono` et `location`.

Le schéma relationnel, où les clés primaires sont soulignées et les clés étrangères sont précédées du symbole #, est le suivant :

`adherent( numero-licence , taille-adherent, nom, prenom)`

`kimono( id-kimono , taille-kimono)`

`location( #numero-licence , #id-kimono , debut, fin)`

L'attribut `id-kimono` est un nombre entier. Les attributs `taille-adherent` et `taille-kimono` sont des nombres entiers dont l'unité est le centimètre et qui sont tous multiples de 10 (100, 110, 120, 130,…).

Les attributs `debut` et `fin` sont des dates au format chaîne de caractères 'AAAA-MM-JJ'. Tous les attributs doivent être renseignés et valides mais l'attribut `fin` peut éventuellement être égal à la chaîne de caractères vide `''` pour les kimonos en cours de location.

On rappelle qu'en langage SQL la fonction d'agrégation `COUNT` permet de compter un nombre d'enregistrements. Par exemple, pour déterminer le nombre d'adhérents du club, on peut utiliser la requête suivante :

```
SELECT COUNT(numero-licence) FROM adherent
```

1. Écrire une requête SQL permettant de connaître le numéro des kimonos **en cours de location**.

2. Écrire une requête SQL permettant de connaître le nombre de kimonos de taille 130 cm possédés par le club.

3. Écrire la requête SQL permettant de connaître le nom et le prénom de l'adhérent qui possède le kimono 42 uniquement si ce kimono est en cours de location. Cette requête ne doit renvoyer que le nom et le prénom de l'adhérent à qui il est actuellement loué et pas celui de tous ceux qui l'ont éventuellement loué par le passé.

Dans une requête SQL, il est possible de modifier un attribut entier avec une expression du type `a = a + 5`.

4. À la fin de l'année, pour anticiper les locations de l'année à venir, le club de judo modifie arbitrairement la taille de tous ses adhérents mesurant strictement moins de 160 cm en leur rajoutant 10 cm. Écrire une requête SQL permettant de réaliser cette opération.

5. Le kimono numéro 25 a été déchiré lors d'un combat. Écrire les requêtes SQL permettant de le supprimer de la base de données.

## Partie B

Un adhérent du club de judo est décrit par les descripteurs :

`nom` : nom de famille de l'adhérent, la donnée est une chaîne de caractères (on supposera dans l'exercice que tous les noms de famille font plus de cinq caractères) ;

`prenom` : prénom de l'adhérent, la donnée est une chaîne de caractères ;

`annee` : année de naissance de l'adhérent, la donnée est une chaîne composée de 4 caractères ;

`mois` : mois de naissance de l'adhérent, la donnée est une chaîne de deux caractères parmi '0123456789' : '01' (pour janvier) et '12' (pour décembre) ;

`jour` : jour de naissance de l'adhérent, la donnée est une chaîne de deux caractères parmi '0123456789' : '01' (si l'adhérent est né le premier jour du mois) et '31' (si l'adhérent est né le 31 du mois) ;

`sexe` : sexe de l'adhérent, la donnée est un caractère valant soit 'F' si l'adhérent est de sexe féminin, soit 'M' s'il est de sexe masculin.

Pour différencier les pratiquants du judo en France, la fédération française de judo, attribue à chaque pratiquant un « numéro de licence ». Il s'agit d'une chaîne de 16 caractères dont :

- le premier caractère vaut soit 'F' soit 'M' selon le sexe de l'adhérent ;
- les huit caractères suivants correspondent à la date de naissance au format 'JJMMAAAA' ;

- les cinq caractères suivants correspondent aux cinq premières lettres du nom de famille de l'adhérent en majuscules ;
- les deux derniers caractères correspondent à un nombre entre 01 et 99. Ils permettent à la fédération française de judo de différencier des pratiquants, dans le cas rare où ils ont le même sexe, la même date de naissance et les cinq premières lettres de leur nom identiques.

**Exemple :** Clémence et Stéphanie Dupond, sœurs jumelles nées le 03/07/1997, se voient attribuées les numéros de licences respectifs 'F03071997DUPON01' et 'F03071997DUPON02'

6. Déterminer un numéro de licence possible pour Eddie Nirrer né le 12/10/2021.
7. Un adhérent a le numéro de licence 'M23091974MARTI01'. Donner sa date de naissance et un nom de famille possible.

Pour manipuler efficacement sa base d'adhérents, le club de judo implémente une table par un tableau nommé `tab_adherents` contenant des dictionnaires en langage Python. Chaque dictionnaire correspond à un adhérent du club.

Les clés des dictionnaires, communes à tous les dictionnaires, correspondent aux descripteurs utilisés pour cette table et sont `nom`, `prenom`, `annee`, `mois`, `jour`, `sexe`, `numero-licence`.

On donne en illustration les deux premiers éléments de ce tableau de dictionnaires,

```
1 tab_adherents= [
2 {'nom': 'DUPOND', 'prenom' : 'CLEMENCE', 'annee' : '1997',
3  'mois' : '07', 'jour' : '03', 'sexe' : 'F',
4  'numero-licence' : 'F03071997DUPON01'},
5 {'nom': 'DUPOND', 'prenom' : 'STEPHANIE', 'annee' : '1997',
6  'mois' : '07', 'jour' : '03', 'sexe' : 'F',
7  'numero-licence' : 'F03071997DUPON02'},
8 ...]
```

8. Donner, sans justifier, la valeur à laquelle on accède avec l'instruction `tab_adherents[1]['prenom']`.
9. Écrire l'instruction permettant d'obtenir la valeur 'F03071997DUPON01'.

La direction du club souhaite connaître le nombre de ses adhérents nés une année donnée. Elle demande d'écrire une fonction `nombre_adherents` qui prend en entrée un tableau de dictionnaires `table` correspondant aux adhérents et une chaîne de 4 caractères `annee` correspondant à une année et qui renvoie en sortie l'entier correspondant au nombre d'adhérents nés cette année.

10. Recopier et compléter la fonction suivante pour permettre de répondre à cette problématique.

```
def nombre_adherents(table, annee):
    compteur = ...
    for adherent in table:
        if ...:
            ...
    ...
```

La direction souhaite également connaître les adhérents les plus âgés du club. Elle nous demande d'écrire une fonction `adherent_plus_age` qui prend en entrée un tableau de dictionnaires `table` correspondant aux adhérents et qui renvoie en sortie une liste de dictionnaires correspondant aux adhérents les plus âgés du club. La comparaison se restreindra uniquement à l'année de naissance. Il peut donc y avoir un ou plusieurs adhérents qui sont nés la même année et qui sont donc les adhérents les plus âgés. On pourra supposer qu'aucun adhérent n'est né après l'année `'2024'` et qu'il y a toujours au moins un adhérent dans le club. On admet que l'on peut comparer en Python deux chaînes de caractères représentant deux années et que le résultat est le même que si ces années étaient des entiers. Par exemple, on a `tab_adherents[0]['annee'] < '2025'`, car la première adhérente est née en 1997.

11. Écrire en Python cette fonction `adherent_plus_age`.

Pour tester la cohérence de sa base et des erreurs causées par un mauvais enregistrement de numéro de licence, le club souhaite programmer une fonction `verification_licence`. Cette fonction prend en paramètre un dictionnaire `adherent` correspondant à un adhérent du club et renvoie `True` si le numéro de licence de l'adhérent est conforme à son sexe, sa date de naissance et son nom de famille et `False` sinon. On pourra supposer que l'on dispose d'une fonction `extraire` qui prend en paramètre une chaîne de caractères `s` et deux indices `i` et `j` avec `0 <= i <= j <= len(s)` et qui renvoie la sous-chaîne de `s` entre les indices `i` inclus et `j` exclu. Par exemple :

```
>>> no = 'F03071997DUPON01'
>>> extraire(no, 0, len(no))
'F03071997DUPON01'
>>> extraire(no, 5, 9)
'1997'
```

12. Écrire la fonction `verification_licence` en Python.

## **Exercice 2 :** *(6 points)*

*Cet exercice porte sur les listes, les dictionnaires, les fonctions et la récursivité.*

Nous souhaitons créer en langage Python un dictionnaire contenant un grand nombre de mots de façon à ce qu'une recherche dans ce dictionnaire soit la plus rapide possible.

Pour cela nous allons créer des groupes de mots, chaque groupe sera une liste Python associée à une clé unique dans le dictionnaire.

Voici un extrait du dictionnaire que nous souhaitons créer (les `...` indiquent des éléments non listés dans cet extrait) :

```
d = {
    44  : ['ABAISSEMENT', 'ADMINISTRATEUR', ..., 'VERSETS'],
    74  : ['ABAISSER', 'ABLATION', ..., 'TROU'],
    243 : ['ABANDON', 'ALLEGRETTO', ..., 'ZIP'],
    36  : ['ABANDONNANT', 'ABOLITIONNISTE', ..., 'VOULAIT'],
    134 : ['ABANDONNE', 'AGNOSTICISME', ..., 'VOIES'],
    40  : ['ABANDONNENT', 'ACCOUCHEUSE', ..., 'YACK'],
    ...
}
```

Chaque clé sera un nombre entier positif et chaque valeur sera une liste de mots.

### Partie A

Pour générer ces valeurs de clés, nous utiliserons une fonction dite de *hachage*, c'est-à-dire une fonction qui pour un mot donné calculera la clé qui lui sera associée. Nous choisissons une fonction de *hachage* simple qui consiste à additionner sur un octet les codes ASCII de chaque lettre de ce mot.

Nous n'utiliserons que des lettres majuscules de l'alphabet, nous rappelons ici le code ASCII (en hexadécimal, c'est-à-dire en base 16) de chacune de ces lettres. Chaque valeur en hexadécimal est notée avec le préfixe `0x`, par exemple `0x15` correspond en décimal à $1 \times 16 + 5 \times 1 = 21$.

| Table des codes `ASCII` | | | | | |
|---|---|---|---|---|---|
| 'A' : `0x41` | 'F' : `0x46` | 'K' : `0x4B` | 'P' : `0x50` | 'U' : `0x55` | 'Z' : `0x5A` |
| 'B' : `0x42` | 'G' : `0x47` | 'L' : `0x4C` | 'Q' : `0x51` | 'V' : `0x56` | |
| 'C' : `0x43` | 'H' : `0x48` | 'M' : `0x4D` | 'R' : `0x52` | 'W' : `0x57` | |
| 'D' : `0x44` | 'I' : `0x49` | 'N' : `0x4E` | 'S' : `0x53` | 'X' : `0x58` | |
| 'E' : `0x45` | 'J' : `0x4A` | 'O' : `0x4F` | 'T' : `0x54` | 'Y' : `0x59` | |

Ainsi le mot `'NSI'` aura pour clé `0xEA` puisque : `0x4E` + `0x53` + `0x49` = `0xEA`, ou `234` en décimal.

Si la somme des codes ASCII ne tient pas sur un octet, seul l'octet de poids faible est conservé.

Par exemple : la somme des codes `ASCII` des lettres du mot `'ADMINISTRATEUR'` est de `0x42C`, la clé qui lui sera associée sera donc `0x2C`, égale à `44` en décimal.

1. Trouver la valeur exprimée en hexadécimal de la clé qui sera associée au mot `'EW'`.
2. Comparer, sans les calculer, les valeurs des clés qui seront associées aux mots `'SAC'` et `'CAS'`. Justifier la réponse.

Voici le code de la fonction qui calcule cette clé pour un mot donné (nous rappelons que la fonction `ord` renvoie le code ASCII d'un caractère) :

```
1 def code_hachage(mot):
2     somme = ...
3     for caractere in ...:
4         somme = ...
5     return somme % 0x100
```

3. Recopier et compléter les lignes 2, 3 et 4 de la fonction `code_hachage`.
4. Expliquer l'expression `somme % 0x100` dans le code ci-dessus et donner les valeurs possibles pour la clé.

## Partie B

Les listes de mots associées à chaque clé sont rangées dans l'ordre alphabétique, ce qui facilitera la recherche d'un mot par la suite.

Nous allons maintenant voir l'écriture d'une fonction permettant l'ajout d'un mot dans une liste de mots en maintenant un ordre alphabétique dans cette liste.

Nous rappelons qu'en langage Python la comparaison de chaînes de caractères est possible à l'aide des opérateurs classiques de comparaison : <, >, ==, >= et <= . Ces opérateurs utilisent l'ordre alphabétique.

Par exemple, l'expression booléenne `'ANNEE' < 'BATEAU'` vaut `True` puisque le mot `'ANNEE'` est placé avant le mot `'BATEAU'` dans l'ordre alphabétique.

Voici le code d'une fonction `ajouter_mot_liste(liste, mot)` qui a pour paramètres `liste`, une liste de chaînes de caractères (ce sont les mots) et `mot`, une chaîne de caractères correspondant au mot que l'on souhaite ajouter à `liste`. Cette fonction modifie `liste` en y ajoutant `mot` selon l'ordre alphabétique. Elle renvoie de plus `liste` après cet ajout.

```
def ajouter_mot_liste(liste, mot):
    i = 0
    while i < len(liste):
        if mot < liste[i]:
            # La méthode "insert" permet d'ajouter un
            # élément à un indice donné dans "liste",
            # les éléments suivants sont décalés.
            liste.insert(i, mot)
            return liste
        i = i + 1
    # La méthode "append" permet d'ajouter un nouvel
    # élément à la fin de "liste".
    liste.append(mot)
    return liste
```

Voici des exemples d'utilisation de cette fonction :

```
>>> ajouter_mot_liste([],'NSI')
['NSI']
>>> ajouter_mot_liste(['NSI'],'PYTHON')
['NSI','PYTHON']
>>> ajouter_mot_liste(['NSI','PYTHON'],'OBJET')
['NSI','OBJET','PYTHON']
>>> ajouter_mot_liste(['NSI','OBJET','PYTHON'],'RAM')
['NSI','OBJET','PYTHON','RAM']
```

5. Déterminer, dans le pire cas, l'ordre de grandeur du nombre de comparaisons entre chaînes de caractères effectuées par un appel à `ajouter_mot_liste`. On exprimera le résultat en fonction de $n$, le nombre de mots présents dans `liste`.

Nous souhaitons écrire une fonction qui permette l'ajout d'un mot dans le dictionnaire décrit au début de l'exercice (qui associe à chaque clé entière possible une liste de mots).

6. Rappeler avec quelle expression Python on peut tester si une clé `c` est déjà présente dans un dictionnaire `dico`.

7. Écrire un code pour la fonction `ajouter_mot_dict(dict_mots, mot)` qui a pour paramètres `dict_mots`, un dictionnaire qui associe à chaque clé entière possible une liste de mots, et `mot` le mot à ajouter dans `dict_mots`, et qui réalise cet ajout (mais ne renvoie aucune valeur). On utilisera les fonctions `code_hachage` et `ajouter_mot_liste`.

## Partie C

Nous allons maintenant voir comment faire la recherche d'un mot dans notre dictionnaire.

Nous souhaitons une recherche rapide et nous allons profiter du fait que les mots sont classés dans l'ordre alphabétique dans les listes du dictionnaire.

Nous prendrons une approche dichotomique pour écrire une fonction `est_present(liste, mot, debut, fin)` qui a pour paramètres :

- `liste`, une liste de mots classés dans l'ordre alphabétique ;
- `mot`, le mot à rechercher ;
- `debut` et `fin`, les indices entre lesquels la recherche se fait dans `liste`.

Cette fonction renverra `True` si `liste` contient `mot` entre les indices `debut` (inclus) et `fin` (exclus), et renverra `False` sinon.

Voici son code :

```
 1 def est_present(liste, mot, debut, fin):
 2     if debut > fin:
 3         return False
 4     milieu = (debut + fin) // 2
 5     if liste[milieu] > mot:
 6         return est_present(liste, mot, debut, milieu - 1)
 7     elif liste[milieu] < mot:
 8         return est_present(liste, mot, milieu + 1, fin)
 9     else:
10         return True
```

Nous testons cette fonction ainsi :

```
>>> liste_mots = ['FONCTION', 'NSI', 'PYTHON', 'OBJET', 'RAM']
>>> est_present(liste_mots, 'NSI', 0, len(liste_mots))
True
```

8. Donner, lors de l'exécution de l'exemple précédent, les valeurs des paramètres `debut` et `fin` prises lors de chaque appel de la fonction `est_present`.

9. Expliquer pourquoi la méthode dichotomique permet d'effectuer, en général, moins d'opérations qu'une recherche simple qui consisterait à comparer un par un les mots de la liste avec le mot cherché.

10. Donner l'ordre de grandeur du nombre de comparaisons de mots effectuées par la méthode dichotomique sur une liste de longueur $n$ ?

11. Écrire un code pour la fonction `mot_present(dict_mots, mot)` qui a pour paramètres `dict_mots` , un dictionnaire qui associe à chaque clé entière possible une liste de mots, et `mot`, le mot à rechercher dans `dict_mots`, et qui renvoie `True` si le mot est présent et `False` sinon. On utilisera les fonctions `code_hachage` et `est_present`.

## **Exercice 3 :** *(6 points)*

*Thèmes abordés : structures de données, programmation.*

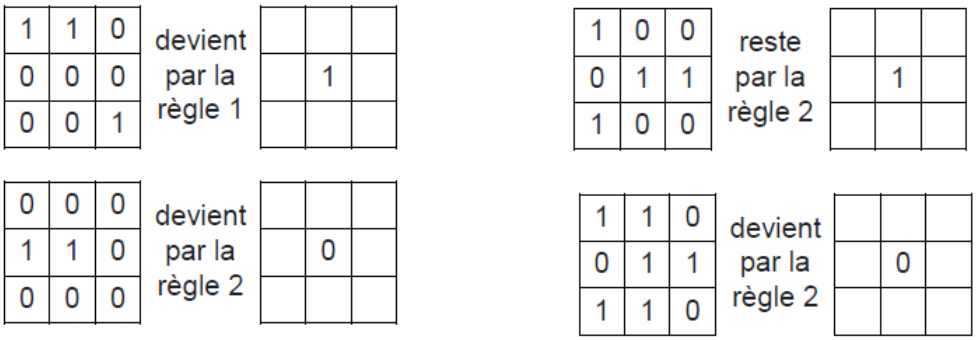
Le « jeu de la vie » se déroule sur une grille à deux dimensions dont les cases, qu'on appelle des « cellules », par analogie avec les cellules vivantes, peuvent prendre deux états distincts : « vivante » (= 1) ou « morte » (= 0).

Une cellule possède au plus huit voisins, qui sont les cellules adjacentes horizontalement, verticalement et diagonalement.

À chaque étape, l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

- Règle 1 : une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît) ; sinon, elle reste à l'état « morte »
- Règle 2 : une cellule vivante possédant deux ou trois voisines vivantes reste vivante, sinon elle meurt.

Voici un exemple d'évolution du jeu de la vie appliquée à la cellule centrale :




Pour initialiser le jeu, on crée en langage Python une grille de dimension **8x8**, modélisée par une liste de listes.

1. Initialisation du tableau :

**a**. Parmi les deux scripts proposés, indiquer celui qui vous semble le plus adapté pour initialiser un tableau de 0. Justifier votre choix

| Choix 1 | Choix 2 |
|---|---|
| `ligne = [0,0,0,0,0,0,0,0]`<br>`jeu = []`<br>`for i in range(8) :`<br>`    jeu.append(ligne)` | `jeu = []`<br>`for i in range(8) :`<br>`    ligne = [0,0,0,0,0,0,0,0]`<br>`    jeu.append(ligne)` |

**b**. Donner l'instruction permettant de modifier la grille `jeu` afin d'obtenir

```
>>> jeu
[[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 1, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0]]
```

2.

**a**. Ecrire en langage Python une fonction `remplissage(n, jeu)` qui prend en paramètres un entier `n` et une grille `jeu`, et qui ajoute aléatoirement exactement `n` cellules vivantes dans le tableau `jeu`.

**b**. Quelles sont les préconditions de cette fonction pour la variable `n` ?

On propose la fonction en langage Python `nombre_de_vivants(i, j, jeu)` qui prend en paramètres deux entiers `i` et `j` ainsi qu'une grille `jeu` et qui renvoie le nombre de voisins **vivants** de la cellule `tab[i][j]` :

```
def nombre_de_vivants(i, j, jeu) :
    nb = 0
    voisins = [(i-1,j-1), (i-1,j), (i-1,j+1), (i,j+1),
               (i+1,j+1), (i+1,j), (i+1,j-1), (i,j-1)]
    for e in voisins :
        if 0 <= ... < 8 and 0 <= ... < 8 :
            nb = nb + jeu[...][...]
    return nb
```

3. Recopier et compléter les pointillés pour que la fonction réponde à la demande.
4. En utilisant la fonction `nombre_de_vivants(i, j, jeu)` précédente, écrire en langage Python une fonction `transfo_cellule(i, j, jeu)` qui prend en paramètres deux entiers `i` et `j` ainsi qu'une grille `jeu` et renvoie le nouvel état de la cellule `jeu[i][j]` (0 ou 1)