Bac Général Maths + NSI (Mathématiques et Numérique et sciences informatiques)

SUJET D'EXAMEN (SESSION OFFICIELLE)

Numérique et sciences informatiques - Session 2026

Sujet de l'examen national tel que publié pour la session indiquée. Source : sujets nationaux publiés par le ministère chargé de l'examen et les services académiques. Document mis à disposition par FormaV pour la révision.

Toutes les annales et leurs corrigés : https://bac-maths-nsi.fr/annales-sujets-et-corriges

BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2026

NUMÉRIQUE ET SCIENCES INFORMATIQUES

JOUR 1

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 1/11 à 11/11.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

EXERCICE 1 (6 points)

Cet exercice porte sur la notion de complexité.

On définit une suite d'entiers U_n de la façon suivante

$$U_0 = 0,\ U_1 = 1,\ U_n = U_{n-1} + U_{n-2} \text{ pour tout } n \geq 2.$$

Les premières valeurs de cette suite sont donc les suivantes : $U_0 = 0$, $U_1 = 1$, $U_2 = 1$, $U_3 = 2$, $U_4 = 3$, $U_5 = 5$.

On donne la fonction suivante permettant de calculer les termes de cette suite :

1. Compléter la fonction récursive `suite` qui prend en paramètre un entier `n` et renvoie la valeur de U_n.

```
1 def suite(n):
2     if n < 2:
3         valeur = ...
4     else:
5         valeur = suite(n - 1) + suite(...)
6     return ...
```

On appelle p la fonction donnant le nombre d'additions à effectuer pour calculer le terme d'indice n de la suite U_n avec la fonction proposée ci-dessus. On peut montrer que cette fonction est la suivante :

$$\begin{cases} p(0) = 0 \\ p(1) = 0 \\ p(n) = 1 + p(n-1) + p(n-2) \text{ pour tout } n \geq 2 \end{cases}$$

2. Écrire une fonction `nb_additions_suite` prenant en paramètre un entier n et renvoyant la valeur de $p(n)$ où p est la fonction définie ci-dessus.
3. Calculer les valeurs *p(2)*, *p(4)* et *p(7).*

On propose une nouvelle fonction pour calculer les termes de la suite U_n :

```
1 def suite2(n, valeurs_calculees = [0, 1]):
2     if n < len(valeurs_calculees):
3         valeur = valeurs_calculees[n]
4     else:
5         valeurs_calculees.append(suite2(n - 1, valeurs_calculees)
                                   + suite2(n - 2, valeurs_calculees))
6     return valeurs_calculees[n]
```

4. Recopier et compléter le tableau ci-dessous avec le nombre d'additions effectuées par la fonction `suite2` pour les calculs des premiers termes de la suite U_n.

complexité de `suite2`	
n	nombre d'additions
2	1
3	2
4	3
5	
6	
7	

On appelle q la fonction donnant le nombre d'additions à effectuer pour calculer le terme d'indice n de la suite U_n avec la fonction `suite2`.

5. Expliquer pourquoi `suite2` est plus efficace que `suite` et donner le nom du procédé algorithmique utilisé pour obtenir cette efficacité.

6. Proposer une version itérative, non récursive, `suite3` du calcul des termes de la suite U_n.

7. Expliquer pourquoi la version itérative est préférable à la version récursive pour obtenir un terme quelconque de la suite de U_n.

EXERCICE 2 (6 points)

Cet exercice porte sur la programmation Python, la Programmation Orientée Objet, les arbres binaires de recherche, les dictionnaires et les traitements sur le fichier CSV.

Dans le cadre d'un projet informatique, des étudiants sont chargés de développer une application console de gestion d'annuaire téléphonique. L'enseignant fournit un fichier contenant des contacts fictifs, chaque contact étant associé à un prénom (clé) et un numéro de téléphone (valeur). Le fichier des contacts est stocké dans la variable ci-dessous.

```
contacts_fictifs = [
    {'Olivier': '01 23 45 67 89'},
    {'Benjamin': '06 12 34 56 78'},
    {'Sophie': '02 34 56 78 90'},
    {'Antoine': '09 87 65 43 21'},
    {'Eric': '03 45 67 89 01'},
    {'Roxanne': '07 45 67 89 01'},
    {'Victor': '03 56 78 90 12'},
    {'Emilie': '04 23 45 67 89'},
    {'Yann': '01 54 32 10 98'},
    {'Zoé': '05 23 45 67 89'}
]
```

Cette application permet d'ajouter, de rechercher des contacts par nom et d'afficher tous les contacts dans l'ordre alphabétique.

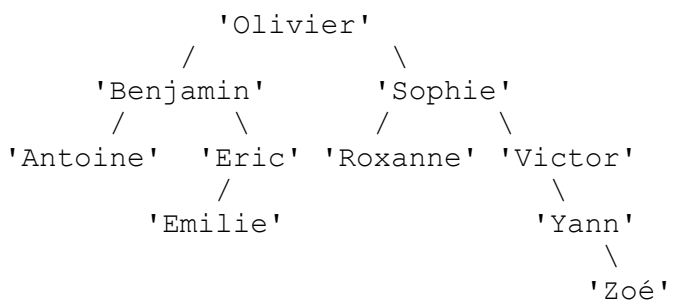
Partie A

Un arbre binaire de recherche est utilisé pour stocker et gérer efficacement ces contacts.

Dans cet exercice, les arbres binaires de recherche ne peuvent pas comporter plusieurs fois la même clé.

Pour un arbre binaire de recherche limité à un nœud, on considère sa **hauteur** comme étant 1.

On considère l'arbre binaire de recherche suivant :



1. Donner la structure de données de la variable `contacts_fictifs`.

2. Justifier que l'arbre binaire ci-dessus est bien un arbre binaire de recherche pour l'ordre alphabétique.
3. Indiquer la racine de cet arbre, puis donner l'ensemble des feuilles de cet arbre.
4. Donner la hauteur et la taille de l'arbre.
5. Citer l'avantage d'utiliser un arbre de recherche par rapport au type de données de la variable `contacts_fictifs`.
6. Donner le sous arbre-droit du nœud `Benjamin`.

On considère les classes suivantes :

```
1  class Contact:
2      def __init__(self, v_prenom, v_num_tel):
3          self.prenom = v_prenom
4          self.num_tel = v_num_tel
5
6  class Noeud:
7      def __init__(self, v_contact, v_gauche=None, v_droite=None):
8          self.contact = v_contact
9          self.gauche = v_gauche
10         self.droite = v_droite
```

7. Parmi les trois instructions (A), (B) et (C) suivantes, écrire sur la copie la lettre correspondant à celle qui construit et stocke dans la variable `abr` le sous-arbre qui a pour racine `Benjamin`.

Instruction A :

```
>>> abr = Noeud(Contact('Benjamin', '06 12 34 56 78'),
Noeud(Contact('Antoine', '09 87 65 43 21')),
Noeud(Contact('Emilie', '04 23 45 67 89'), Noeud(Contact('Eric', '03
45 67 89 01'))))
```

Instruction B :

```
>>> abr = Noeud(Contact('Benjamin', '06 12 34 56 78'),
Noeud(Contact('Antoine', '09 87 65 43 21')),
Noeud(Contact('Eric', '03 45 67 89 01'), Noeud(Contact('Emilie', '04
23 45 67 89'))))
```

Instruction C :

```
>>> abr = Noeud(Noeud(Contact('Benjamin', '06 12 34 56
78'),('Antoine', '09 87 65 43 21')),
Noeud(Contact('Eric', '03 45 67 89 01'), ('Emilie', '04 23 45 67
89')))
```

On considère la classe suivante :

```
1  class ContactManager:
2      def __init__(self):
3          self.root = None
4
5      def inserer_contact(self, contact):
6          self.root = self.inserer_recursive(self.root, contact)
7
8      def inserer_recursive(self, noeud, contact):
9          if noeud is None:
10             return Noeud(contact)
11         if contact.prenom < noeud.contact.prenom:
12             noeud.gauche = self.inserer_recursive(noeud.gauche,
                                                     contact)
13         else:
14             ...
15         return noeud
16
17   def trouver_contact(self, prenom):
18       return self.trouver_recursive(self.root, prenom)
19
20   def trouver_recursive(self, noeud, prenom):
21       if noeud is None:
22           return None
23       ...
24           return noeud.contact.num_tel
25       elif prenom < noeud.contact.prenom:
26           ...
27       else:
28           return self.trouver_recursive(noeud.droite, prenom)
```

8. Donner les instructions à placer dans lignes 14, 23 et 26 pour compléter les méthodes `inserer_recursive` et `trouver_recursive`.

Dans le programme principal, on considère les instructions suivantes :

```
1  # Création du gestionnaire de contacts
2  contact_manager = ContactManager()
3
4  # Insertion des contacts
5  contact_manager.inserer_contact(Contact('Olivier', '01 23 45 67
89'))
6  contact_manager.inserer_contact(Contact('Benjamin', '06 12 34 56
78'))
7  contact_manager.inserer_contact(Contact('Sophie', '02 34 56 78
90'))
8  contact_manager.inserer_contact(Contact('Antoine', '09 87 65 43
21'))
9  contact_manager.inserer_contact(Contact('Eric', '03 45 67 89
01'))
10 contact_manager.inserer_contact(Contact('Roxanne', '07 45 67 89
01'))
11 contact_manager.inserer_contact(Contact('Victor', '03 56 78 90
12'))
```

```
12 contact_manager.inserer_contact(Contact('Emilie', '04 23 45 67
89'))
13 contact_manager.inserer_contact(Contact('Yann', '01 54 32 10
98'))
14 contact_manager.inserer_contact(Contact('Zoé', '05 23 45 67 89'))
```

9. En utilisant la variable `contact_manager` définie à la ligne 2, et en particulier sa méthode `trouver_contact` (de la classe `ContactManager`), écrire en langage Python une fonction `recherche_contact` qui prend en paramètre `prenom_recherche` une chaîne de caractère à rechercher et qui affichera dans la console les résultats suivants en fonction des cas :

```
>>> recherche_contact("Roxanne")
Prénom recherché : Roxanne
Roxanne : 07 45 67 89 01
```

```
>>> recherche_contact("Adrien")
Prénom recherché : Adrien
Contact 'Adrien' introuvable
```

Dans le programme principal, on considère les instructions suivantes :

```
1 print("Liste des contacts triés :")
2 affichage_contacts_trie(contact_manager.root)
```

On souhaite afficher dans la console, la liste des contacts dans l'ordre alphabétique sous la forme suivante :

```
>>>
Liste des contacts triés :
Antoine : 09 87 65 43 21
Benjamin : 06 12 34 56 78
Emilie : 04 23 45 67 89
Eric : 03 45 67 89 01
Olivier : 01 23 45 67 89
Roxanne : 07 45 67 89 01
Sophie : 02 34 56 78 90
Victor : 03 56 78 90 12
Yann : 01 54 32 10 98
Zoé : 05 23 45 67 89
```

10. Indiquer, en le justifiant, le type de parcours à utiliser pour afficher la liste des prénoms dans l'ordre alphabétique (Parcours en largeur - parcours en profondeur dans l'ordre préfixe - parcours en profondeur dans l'ordre infixe - parcours en profondeur dans l'ordre postfixe).

11. Écrire les instructions des lignes 4 et 5 de la fonction récursive `affichage_contacts_trie` afin d'obtenir ce résultat.

```
1 def affichage_contacts_trie(noeud):
2     """Afficher les contacts triés par nom"""
3     if noeud is not None:
4         ...
5         ...
6         affichage_contacts_trie(noeud.droite)
```

Exercice 3 (8 points)

Cet exercice porte sur le paradigme 'Programmation orientée objet`, les bases de données et les arbres binaires de recherche

Introduction:
Au cœur du développement technologique, les interfaces conversationnelles, ou chatbots, jouent un rôle de plus en plus important dans l'interaction avec les utilisateurs. La start-up OpenChat souhaite développer un chatbot qui utilise une base de données pour stocker des questions et les réponses associées.

Partie 1 : Structures de données

On souhaite créer une classe `Chatbot` ayant un attribut `base_donnees` initialisé comme un dictionnaire vide.

On utilisera ensuite l'attribut `base_donnees` qui permettra de stocker les questions en tant que clés du dictionnaire et les réponses à ces questions en tant que valeurs associées à ses clés.

On définit également deux méthodes permettant de faire fonctionner le chatbot.

- Une méthode `ajouter_question_reponse` permet l'ajout dans le dictionnaire `base_donnees` de la valeur `reponse` associée à la clé `question`.
- Une méthode `repondre` renvoie la réponse associée à une question.

1. Recopier et compléter les lignes 3 et 6 dans le constructeur de la classe `Chatbot` et dans la méthode `ajouter_question_reponse`.

```
1    class Chatbot:
2        def __init__(self):
3            self.base_donnees = ...   # à compléter
4
5        def ajouter_question_reponse(self, question, reponse):
6            self.base_donnees[question] = ...  # à compléter
7
8        def repondre(self, question):
9            return self.base_donnees[question]
```

Nous allons maintenant illustrer quelques interactions possibles entre un utilisateur et le chatbot, en se basant sur la classe précédemment définie.

2. Écrire les lignes de code permettant d'instancier un objet `Bot1` de la classe `Chatbot` et d'ajouter à son dictionnaire l'interaction :

Question : "Bonjour, comment vas-tu ?" Réponse : "Bonjour, je vais bien, merci."

3. Écrire les lignes de code permettant d'obtenir la réponse à la question "Bonjour, comment vas-tu ?"

Partie 2 : Interaction avec la base de données

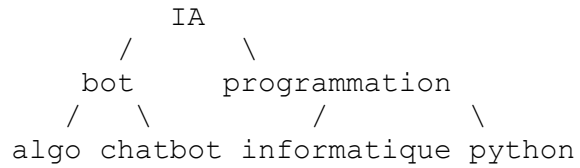
Cette partie porte sur les bases de données. On pourra utiliser les mots clés SQL suivants : AND, FROM, INSERT, INTO, JOIN, OR, ON, SELECT, SET, UPDATE, VALUES, WHERE.

On considère une table `questions_reponses` avec les colonnes `id`, `question`, `reponse` et `mot_cle`.

4. Associer à chaque mot clé sa définition :
   - Mots clés : SELECT - INSERT - UPDATE
   - Définitions : ajouter une donnée - extraire des données - mettre à jour une table
5. Définir ce qu'est une *clé primaire*.
6. Indiquer quel attribut (éventuellement plusieurs) serait le plus adapté pour être une clé primaire dans la table `questions_reponses` et justifier votre choix.
7. Écrire une requête permettant de récupérer toutes les questions comportant comme mot-clé "python".
8. Écrire une requête pour insérer la nouvelle question-réponse « Date de la bataille de Marignan ? », « 1515 » associé au mot clef « Histoire ». On prendra comme identifiant Id=15.
9. Écrire la requête pour mettre à jour la réponse à la question d'Id = 12 qui devient « Oui ».

Partie 3 : Représentation de la recherche avec les arbres

On considère cet exemple d'un Arbre Binaire de Recherche (ABR) basé sur le mot-clé des questions :



10. Indiquer quel type de parcours donnerait, dans cet ordre : IA, bot, programmation, algo, chatbot, informatique, python.
11. Indiquer dans quel ordre apparaitraient les mots-clés avec un parcours en profondeur infixe sur l'ABR illustré.
12. Indiquer dans quel ordre apparaitraient les mots-clés pour un parcours en profondeur postfixe (ou suffixe).
13. Indiquer quel parcours permettrait d'obtenir les mots-clés dans l'ordre alphabétique.

14. Compléter la méthode de recherche dans l'ABR :

```
class Noeud:

    def __init__(self, cle):
        self.cle = cle
        self.gauche = None
        self.droit = None

    def recherche(arbre, cle):
        if arbre is None or arbre.cle == ... : # à compléter
            return arbre
        if cle < arbre.cle:
            return recherche(arbre. ..., cle) # à compléter
        else:
            return recherche(arbre. ..., cle) # à compléter
```