Corrigé - NSI Amérique du Nord Jour 1 2026

Ce sujet NSI Amérique du Nord Jour 1 2026 couvre parfaitement les thèmes du programme de Terminale NSI : programmation orientée objet, récursivité, algorithme Min-Max, réseaux (RIP, OSPF, TCP/IP), files d'attente, SQL et programmation dynamique.

Exercice 1 (6 points)

Programmation orientée objet – Récursivité – Min-Max

Question 1

On doit créer une grille de 6 lignes et 7 colonnes remplie de zéros.

Correction

```
def __init__(self):
    self.grille = [[0 for j in range(7)] for i in range(6)]
```

Explication

Chaque case vide contient 0.

La compréhension de liste permet de construire :

```
[
 [0,0,0,0,0,0,0],
 [0,0,0,0,0,0,0],
 ...
]
```

Question 2

Compléter la méthode joue.

Correction

```
def joue(self, colonne, joueur):
    ligne = 5

    while ligne != -1 and self.grille[ligne][colonne] != 0:
        ligne = ligne - 1

    if ligne != -1:
        self.grille[ligne][colonne] = joueur
        return True
    else:
        return False
```

Lignes attendues

```
4 : ligne = ligne - 1
6 : joueur
7 : True
9 : False
```

Ce que vérifie le correcteur

Comprendre que :

- on part du bas ;
- on remonte ;
- dès qu'on trouve une case vide on place le pion.

Question 3

Créer la grille donnée.

Position des pions :

```
joueur 1 :
(5,2)
(4,3)

joueur 2 :
(5,3)
```

Correction

```
jeu1 = Grille()

jeu1.joue(2,1)
jeu1.joue(3,2)
jeu1.joue(3,1)
```

Pourquoi ?

Le premier pion du joueur 1 tombe ligne 5 colonne 2.
Le joueur 2 joue ensuite colonne 3.
Enfin le joueur 1 joue encore colonne 3.
Le pion se place alors au-dessus.

Question 4

Calcul du score.
On additionne :

- valeurs des pions du joueur 2
- puis on soustrait celles du joueur 1

On suppose :

```
(5,2) = valeur_case(5,2)
(5,3) = valeur_case(5,3)
(4,3) = valeur_case(4,3)
```

Le score vaut :

```
score =
valeur_case(5,3)
-
valeur_case(5,2)
-
valeur_case(4,3)
```

Question 5

Correction

```
def score(self):

    s = 0

    for ligne in range(6):
        for colonne in range(7):

            if self.grille[ligne][colonne] == 1:
                s = s - valeur_case(ligne,colonne)

            elif self.grille[ligne][colonne] == 2:
                s = s + valeur_case(ligne,colonne)

    return s
```

Point pédagogique

Les pions du joueur 1 comptent négativement.
Les pions du joueur 2 comptent positivement.

Question 6

Constructeur de Noeud.

Correction

```
class Noeud:

    def __init__(self,colonne):

        self.colonne = colonne
        self.score = 0
        self.suivants = []
```

Question 7

Méthode colonne_score_min.

Correction

```
def colonne_score_min(self):

    meilleur = self.suivants[0]

    for noeud in self.suivants:

        if noeud.score < meilleur.score:
            meilleur = noeud

    return (meilleur.colonne, meilleur.score)
```

Question 8

Méthode calcule_score.

Lignes attendues

```
4  : -(100 + 10*(niveau_max-niveau))

6  : 100 + 10*(niveau_max-niveau)

8  : grille.score()

13 : Noeud(colonne)

14 : nouveau_noeud

15 : niveau+1,
     3-joueur,
     grille2

17 : self.colonne_score_min()[1]

19 : self.colonne_score_max()[1]
```

Explication fondamentale

Le principe Min-Max :

Joueur 1 :
`cherche le minimum`
Joueur 2 :
`cherche le maximum`
C'est le cœur de l'intelligence artificielle utilisée dans les jeux.

Question 9
Pourquoi niveau_max = 42 est irréaliste ?

Réponse attendue

Chaque coup peut produire jusqu'à :
`7 nouveaux coups`
L'arbre contient donc environ :
`7^42`
possibilités.
Ce nombre est gigantesque.
Le temps de calcul serait beaucoup trop long.

Question 10
Fonction choisit_coup.

Correction

```
def choisit_coup(grille,joueur):

    racine = Noeud(-1)

    racine.calcule_score(
        0,
        joueur,
        grille
    )

    if joueur == 1:
        return racine.colonne_score_min()[0]

    else:
        return racine.colonne_score_max()[0]
```

Bilan Exercice 1
Notions évaluées :

- POO ;
- listes ;
- récursivité ;
- arbre ;
- IA Min-Max.

Ces notions figurent explicitement parmi les compétences NSI attendues au cycle terminal : algorithmique, programmation Python, structures de données et résolution de problèmes.

Exercice 2 — Réseaux, files et POO

Cet exercice porte sur les réseaux, les protocoles RIP/OSPF, l'encapsulation TCP/IP et les files d'attente dans un routeur. Le sujet précise que l'entreprise Gamerzz possède plusieurs sous-réseaux en notation CIDR.

1. Réseau Administration

Le réseau Administration est :

`10.42.0.80/29`

Un réseau `/29` contient :

`2^(32-29) = 2^3 = 8 adresses`

Mais deux adresses ne sont pas attribuables :

- l'adresse réseau ;
- l'adresse de diffusion.

Donc :

`8 - 2 = 6 adresses attribuables`

Adresse réseau :

`10.42.0.80`

Adresse de diffusion :

`10.42.0.87`

Adresses utilisables :

`10.42.0.81 à 10.42.0.86`

Une adresse possible pour le routeur E est donc :

`10.42.0.81`

2. Réseau de la machine 10.42.0.70

Le réseau **Salle Gaming Online** est :

`10.42.0.64/28`

Un `/28` contient 16 adresses :

`10.42.0.64 à 10.42.0.79`

Donc l'adresse :

`10.42.0.70`

appartient au réseau :

`Salle Gaming Online`

3. Adresses des routeurs dans les réseaux /30

Un réseau `/30` contient 4 adresses :

- adresse réseau ;
- deux adresses utilisables ;
- adresse de diffusion.

Réseau 10.42.0.16/30

Adresses :

```
10.42.0.16 : réseau
10.42.0.17 : utilisable
10.42.0.18 : utilisable
10.42.0.19 : diffusion
```

Les routeurs sont C et F, dans l'ordre alphabétique.

Donc :

```
C : 10.42.0.17
F : 10.42.0.18
```

Réseau 10.42.0.12/30

Adresses :

```
10.42.0.12 : réseau
10.42.0.13 : utilisable
10.42.0.14 : utilisable
10.42.0.15 : diffusion
```

Les routeurs sont C et D.

Donc :

```
C : 10.42.0.13
D : 10.42.0.14
```

4. Table RIP possible du routeur C

Le protocole RIP minimise le nombre de routeurs traversés. Le sujet donne la table de routage du routeur B comme modèle.

Table possible pour C :

Réseau	Passerelle	Nombre de sauts
Gaming Online	connecté	0
Gaming VR	10.42.0.18	1
DMZ	10.42.0.1	1
Internet	10.42.0.1	2
Administration	10.42.0.1	2
Application	10.42.0.14	1
SGBD	10.42.0.14	2

Pour **Administration**, on pourrait aussi passer par D puis E selon le chemin choisi, car RIP accepte parfois plusieurs chemins de même coût. Une table possible suffit.

5. Coût OSPF des liaisons

La formule donnée est :

```
coût = 10^10 / débit
```

Les débits visibles sont :

```
10 Gb/s = 10^10 bits/s
1 Gb/s = 10^9 bits/s
100 Mb/s = 10^8 bits/s
```

Donc :

Débit	Coût
10 Gb/s	1
1 Gb/s	10
100 Mb/s	100

6. Chemin OSPF de A vers Application

Le réseau Application est relié au routeur D.

Depuis A, on passe d'abord par B.

Chemins possibles :

```
A → B → D
```

Coût :

```
A-B = 1
B-D = 100
Total = 101
```

Mais on peut faire mieux :

```
A → B → C → D
```

Coût :

```
A-B = 1
B-C = 10
C-D = 1
Total = 12
```
Ou encore :
```
A → B → E → D
```
Coût :
```
A-B = 1
B-E = 10
E-D = 1
Total = 12
```
Réponse possible :
```
A → B → C → D
```
ou :
```
A → B → E → D
```

7. Encapsulation TCP/IP

Les données d'un segment TCP sont contenues dans un paquet IP.
On dit que :
```
TCP est encapsulé dans IP.
```
Donc la bonne formulation est :
```
Un segment TCP est transporté dans un paquet IP.
```

8. Pourquoi utiliser une file plutôt qu'une pile ?

Une **file** fonctionne en FIFO :
```
First In, First Out
```
Le premier paquet arrivé est le premier transmis.
C'est logique pour un routeur, car cela respecte l'ordre d'arrivée et évite qu'un paquet ancien reste bloqué trop longtemps.
Une **pile** fonctionne en LIFO :
```
Last In, First Out
```
Le dernier paquet arrivé serait traité en premier, ce qui pourrait retarder indéfiniment les paquets les plus anciens.
Donc une file est plus juste et plus adaptée au traitement des paquets réseau.

9. Classe Routeur_DROP_TAIL — méthode `__init__`

Le routeur doit mémoriser :

- une file vide ;
- la taille maximale ;
- la taille actuelle, initialement 0.

Correction :

```
class Routeur_DROP_TAIL:

    def __init__(self, t_max):
        self.f = cree_file()
        self.t_max = t_max
        self.t = 0
```

10. Méthode `recoit` pour DROP TAIL

Un paquet est accepté si :
```
t < t_max
```
Correction :

```
def recoit(self, p):
    if self.t < self.t_max:
        enfile(self.f, p)
        self.t = self.t + 1
        return True
    return False
```

Explication :

- si la file n'est pas pleine, on ajoute le paquet ;
- sinon, on le rejette.

11. Méthode `recoit` pour Routeur_ALEA

Règles :

- si `t < t_min`, le paquet est accepté ;
- si `t_min <= t < t_max`, tirage au sort ;
- si `t >= t_max`, rejet.

Correction :

```
def recoit(self, p):
    if self.t < self.t_min:
        enfile(self.f, p)
        self.t = self.t + 1
        return True

    elif self.t_min <= self.t < self.t_max:
        if self.tirage_au_sort():
            enfile(self.f, p)
            self.t = self.t + 1
            return True
        return False

    return False
```

Ce que le correcteur valorise

Il faut bien distinguer les trois cas :

```
file peu remplie → acceptation
file moyennement remplie → rejet aléatoire possible
file pleine → rejet systématique
```

Exercice 3 — SQL, récursivité et programmation dynamique

L'exercice porte sur une base de données immobilière, puis sur la plus longue sous-séquence strictement croissante.

Partie A — Bases de données SQL

1. Pourquoi le numéro dans la rue n'est-il pas une clé primaire ?

Le numéro d'un immeuble dans une rue n'est pas forcément unique dans toute la base.

Par exemple, il peut exister :

```
13 rue Turing
13 rue de la mer
13 rue Ada Lovelace
```

Donc `numero_immeuble` seul ne permet pas d'identifier un immeuble de manière unique.

On utilise donc :
`id_immeuble`
comme clé primaire.

2. Identifiants des immeubles de la rue `'la mer'`

Correction :

```
SELECT id_immeuble
FROM immeuble
WHERE rue_immeuble = 'la mer'
ORDER BY id_immeuble;
```

3. Appartements de l'immeuble 16 au moins au 5e étage

Correction :

```
SELECT id_appart
FROM appartement
WHERE id_immeuble = 16
AND etage_appart >= 5;
```

4. Pourquoi supprimer l'immeuble 16 peut rompre l'intégrité ?

La requête est :

```
DELETE FROM immeuble
WHERE id_immeuble = 16;
```

Problème : des appartements peuvent encore faire référence à cet immeuble avec la clé étrangère :
`appartement.id_immeuble`
Si on supprime l'immeuble sans supprimer ou modifier les appartements associés, certains appartements feront référence à un immeuble qui n'existe plus.
Cela rompt l'intégrité référentielle.
Réponse attendue :

```
Il faut d'abord supprimer les appartements de l'immeuble 16, ou utiliser une
suppression en cascade si elle est prévue.
```

5. Ajouter l'immeuble 140

Immeuble :

- identifiant : 140 ;
- 6 étages ;
- numéro 13 ;
- rue `'Turing'`.

Correction propre :

```
INSERT INTO immeuble
(id_immeuble, nb_etage_immeuble, numero_immeuble, rue_immeuble)
VALUES
(140, 6, 13, 'Turing');
```

6. Doubler le prix de l'appartement 603

Correction :

```
UPDATE appartement
SET prix_appart = 2 * prix_appart
WHERE id_appart = 603;
```

7. Prix maximal d'un appartement dans la rue `'la mer'`

Il faut joindre les deux tables.
Correction :

```
SELECT MAX(prix_appart)
FROM appartement
```

```
JOIN immeuble
ON appartement.id_immeuble = immeuble.id_immeuble
WHERE rue_immeuble = 'la mer';
```

Explication

On a besoin de :

- `prix_appart` dans la table `appartement` ;
- `rue_immeuble` dans la table `immeuble`.

Donc il faut un `JOIN`.

Partie B — Plus longue sous-séquence strictement croissante

On travaille avec :

```
L2 = [3, 1, 8, 2, 5]
```

8. Sous-séquences strictement croissantes de longueur 2

On garde l'ordre des éléments.

Liste complète :

```
[3, 8]
[3, 5]
[1, 8]
[1, 2]
[1, 5]
[2, 5]
```

Attention :

```
[8, 5]
```

n'est pas croissante.

Et :

```
[2, 1]
```

n'est pas une sous-séquence valide dans cet ordre.

9. Plus longue sous-séquence strictement croissante de L2

La plus longue est :

```
[1, 2, 5]
```

Elle est de longueur :

```
3
```

10. Fonction `est_strict_croissante`

Correction :

```
def est_strict_croissante(seq):
    for i in range(len(seq) - 1):
        if seq[i] >= seq[i + 1]:
            return False
    return True
```

Explication

Une liste est strictement croissante si chaque terme est strictement inférieur au suivant.

11. Fonction récursive `llsc_fin`

Code à compléter :

```
def llsc_fin(tab, i):
    if ...:
        return ...
```

```
    max_len = 1
    for j in range(i):
        if tab[j] < ...:
            max_len = max(max_len, llsc_fin(tab, j)+1)
    return max_len
```

Correction :

```
def llsc_fin(tab, i):
    if i == 0:
        return 1

    max_len = 1

    for j in range(i):
        if tab[j] < tab[i]:
            max_len = max(max_len, llsc_fin(tab, j) + 1)

    return max_len
```

Lignes attendues

```
2 : i == 0
3 : 1
6 : tab[i]
```

Explication pédagogique

`llsc_fin(tab, i)` calcule la longueur maximale d'une sous-séquence strictement croissante qui se termine à l'indice `i`.

Si `tab[j] < tab[i]`, alors on peut prolonger une sous-séquence qui se termine en `j` avec l'élément `tab[i]`.

12. Programmation dynamique `llsc_dyn`

Code à compléter :

```
def llsc_dyn(tab):
    n = len(tab)
    dyn = [1] * n
    for i in range(1, n):
        for j in range(i):
            if tab[j] < tab[i]:
                dyn[i] = max(..., ...)
    return ...
```

Correction :

```
def llsc_dyn(tab):
    n = len(tab)
    dyn = [1] * n

    for i in range(1, n):
        for j in range(i):
            if tab[j] < tab[i]:
                dyn[i] = max(dyn[i], dyn[j] + 1)

    return max(dyn)
```

Explication

`dyn[i]` contient la longueur de la plus longue sous-séquence strictement croissante qui se termine à l'indice `i`.

Pour `L2 = [3, 1, 8, 2, 5]`, on obtient :
`dyn = [1, 1, 2, 2, 3]`
Donc :
`max(dyn) = 3`

13. Avantage de la programmation dynamique

La programmation dynamique évite de recalculer plusieurs fois les mêmes sous-problèmes.
La version récursive peut refaire de nombreux appels identiques.
La version dynamique est donc :

- plus efficace ;
- plus rapide ;
- plus stable pour de grandes listes ;
- moins exposée aux limites de profondeur de récursion.

Réponse attendue :
`La programmation dynamique mémorise les résultats intermédiaires et évite les recalculs inutiles.`

Bilan global

Pour obtenir la totalité des points sur ce sujet NSI, il fallait :

- écrire du code Python simple, lisible et correctement indenté ;
- bien respecter les conventions du sujet ;
- justifier les calculs CIDR ;
- distinguer RIP et OSPF ;
- comprendre FIFO/LIFO ;
- maîtriser les clés primaires et étrangères en SQL ;
- savoir écrire une jointure ;
- comprendre la récursivité ;
- expliquer l'intérêt de la programmation dynamique.

Le sujet mobilise exactement les grands piliers de la spécialité NSI : données, algorithmes, langages, machines et réseaux. Le programme insiste notamment sur la conception algorithmique, la programmation, les réseaux, les structures de données et les bases de la démarche informatique.