Bac blanc NSI — décembre 2025

Corrigé

Bruno DARID — Décembre 2025

Plan

Corrigé détaillé – Exercice 1 (Terminale NSI - décembre 2025)

> Version avec *adaptation* de la question 8

> Barème sur 60 ramené ensuite sur 6 points

Question	Points
Q1	10 pts
Q2	10 pts
Q3	4 pts
Q4	10 pts
Q5	10 pts
Q6	2 pts
Q7	6 pts
Q8	8 pts
Total	**60 pts**

Q1 – Fonction `possible_avec_penalites_seules(score)` (10 pts)

```
def possible_avec_penalites_seules(score):
    if score < 0:
        return False
    return score % 3 == 0
```

Q2 – Tableau des scores de 0 à 10 (10 pts)

Score	Solutions possibles	Nombre
0	[0]	1
1	[]	0
2	[]	0
3	[0,3]	1
4	[]	0
5	[0,5]	1
6	[0,3,6]	1
7	[0,7]	1
8	[0,5,8] ; [0,3,8]	2
9	[0,3,6,9]	1
10	[0,3,10] ; [0,5,10] ; [0,7,10]	3

Q3 – Vérification de la relation (4 pts)

$f(10) = f(7) + f(5) + f(3) = 1 + 1 + 1 = 3$

Q4 – Cas de base (10 pts)

n	f(n)
0	1
1	0
2	0
3	1
4	0
5	1
6	1

Q5 – Fonction récursive optimisée (10 pts)

```
def nb_solutions(n):
    if n < 0:
        return 0
    if n == 0:
        return 1
    return nb_solutions(n - 3) + nb_solutions(n - 5) + nb_solutions(n - 7
```

Q6 – Optimisation (2 pts)

On peut utiliser la **mémoïsation** ou la **programmation dynamique** pour stocker les résultats déjà calculés et éviter les recalculs.

Q7 – Lignes utiles pour score égal à 11 (6 pts)

On utilise les lignes correspondant aux scores :

- 11 − 3 = 8
- 11 − 5 = 6
- 11 − 7 = 4

Q8 – Fonction `solutions_possibles(score)` (8 pts)

Important : problème d'indentation dans l'énoncé

Dans l'énoncé original, l'indentation de la ligne `resultat.append(...)` est ambiguë.
Pour correspondre à l'objectif du problème (« construire la liste de **toutes** les solutions possibles »), cette instruction doit être placée **dans la boucle sur les solutions**.

La version fonctionnelle attendue est donc :

```
def solutions_possibles(score):
    if score < 0:
        return []
    if score == 0:
        return [[0]]
    resultat = []
    for coup in [3, 5, 7]:
        liste = solutions_possibles(score - coup)
        for solution in liste:
            nouvelle = solution.copy()
            nouvelle.append(score)
            resultat.append(nouvelle)
    return resultat
```

Barème Q8 (8 pts)

- (2 pts) Cas `score < 0` → `[]`
- (2 pts) Cas `score == 0` → `[[0]]`
- (2 pts) Boucle sur les coups `[3,5,7]` et appel récursif
- (2 pts) Construction correcte des solutions complètes

Corrigé détaillé – Exercice 2 « Tour de France » (Terminale NSI décembre 2025)

> Python (dictionnaires / listes) et SQL – Barème sur 100

> Barème proposé (sur 100)

Partie A – Python (50 pts)

- **Q1** (3 accès demandés) : **14 pts**
- **Q2** (`calcul_temps_total`) : **20 pts**
- **Q3** (mise à jour `classement_general`, insertion triée) : **10 pts**
- **Q4** (construction `tableau_final`) : **6 pts**

Total Partie A : 50 pts

Q1 (14 pts) – Accès dans les dictionnaires/listes

On dispose de :

- `participants` : *nomCoureur* → *numDossard*
- `temps_etapes` : *numDossard* → liste des temps (en secondes) par étape
- `classement_general` : *numDossard* → rang au classement général

a) Numéro de dossard de `PHILIPSEN Jasper`

```
participants["PHILIPSEN Jasper"]
```

b) Classement général de `PHILIPSEN Jasper`

```
classement_general[participants["PHILIPSEN Jasper"]]
```

c) Temps (en secondes) de `PINOT Thibaut` à la 4e étape
(4e étape → indice 3 dans une liste indexée à partir de 0)

```
temps_etapes[participants["PINOT Thibaut"]][3]
```

Barème Q1 (14 pts) - 5 pts par item (clé correcte + accès correct) ; 4 pts pour indice correct

---

Q2 (20 pts) – Fonction `calcul_temps_total(d)`

Objectif : renvoyer le temps total (en secondes) du coureur de dossard `d` en additionnant ses temps d'étapes.

```
def calcul_temps_total(d):
    """Renvoie le temps total (en secondes) du coureur de dossard d."""
    total = 0
    for t in temps_etapes[d]:
        total += t
    return total
```

> Variante acceptable (si autorisée) : `return sum(temps_etapes[d])`.

Barème Q2 (20 pts)

- 4 pts : en-tête correct (nom, paramètre, valeur renvoyée)
- 7 pts : somme correcte des valeurs de `temps_etapes[d]` (boucle + accumulateur ou `sum`)
- 7 pts : renvoi du résultat
- 2 pts : lisibilité (docstring/commentaire, noms pertinents)

Q3 (10 pts) – Compléter l'algorithme de classement (tri par insertion 1re)

Le programme construit une liste `classement` de tuples `(dossard, temps_total)` et l'insère **triée par temps total croissant** (principe d'un tri par insertion).
Compléter les lignes 8, 9, 14.

```
classement = []
for numero_dossard in temps_etapes:
    element = (numero_dossard, calcul_temps_total(numero_dossard))
    classement.append(element)
    pos = len(classement) - 2
    while pos >= 0 and element[1] < classement[pos][1]:
        classement[pos + 1] = classement[pos]
        pos = pos - 1
    classement[pos + 1] = element
for i in range(len(classement)):
    classement_general[classement[i][0]] = i + 1
```

Barème Q3 (10 pts)

- 4 pts : condition du `while` correcte (`element[1]` et `classement[pos][1]`)
- 3 pts : décalage correct (`classement[pos + 1] = classement[pos]`)
- 3 pts : mise à jour correcte de `classement_general` (bonne clé + bon rang `i+1`)

Explications supplémentaires

> **Lecture globale** (ce que fait l'algorithme)

👉 On construit progressivement une liste classement qui contient des couples :

`(dossard, temps_total)`

et cette liste est toujours triée par temps total croissant. C'est exactement le principe du **tri par insertion**: on insère chaque nouvel élément à la bonne place dans une liste déjà triée.

> **Lecture ligne par ligne**

1️⃣ Initialisation `classement = []`

On crée une liste vide qui va contenir le classement général.

2️⃣ Parcours de tous les coureurs `for numero_dossard in temps_etapes:`

On parcourt tous les dossards (les clés du dictionnaire `temps_etapes`).

3️⃣ Construction du couple (dossard, temps_total)
`element = (numero_dossard, calcul_temps_total(numero_dossard))`

On crée un tuple : `(dossard, temps_total)`

Exemple :

`(12, 34567)`

4️⃣ Ajout provisoire en fin de liste `classement.append(element)`

On ajoute le nouvel élément à la fin, même s'il n'est pas encore à la bonne place.

5️⃣ Position de départ pour l'insertion `pos = len(classement) - 2`

Pourquoi -2 ?

Parce que le nouvel élément vient d'être ajouté à la fin et on veut comparer avec l'élément juste avant

Exemple :

```
classement = [(3,1200), (7,1500), (12,1300)]
                                ↑ nouvel élément
```

On démarre la comparaison avec l'indice précédent.

6️⃣ Décalage vers la droite tant que l'ordre n'est pas bon

`while pos >= 0 and element[1] < classement[pos][1]:`

Tant que :

- on n'est pas au début de la liste
- le temps du nouvel élément est plus petit que celui de l'élément précédent

➡️ alors on décale l'élément précédent vers la droite.

7️⃣ Décalage `classement[pos + 1] = classement[pos]`

On décale l'élément vers la droite pour faire de la place.

8️⃣ On recule d'une position `pos = pos - 1`

On continue à comparer plus à gauche.

9️⃣ Insertion définitive à la bonne place `classement[pos + 1] = element`

Quand on sort de la boucle, `pos+1` est la bonne position pour insérer l'élément.

🔟 Construction du classement général

```
for i in range(len(classement)):
    classement_general[classement[i][0]] = i + 1
```

On transforme la liste triée en dictionnaire : dossard → rang

Car :

`classement[i][0] = dossard`

`i+1 = rang` (on commence à 1, pas à 0)

Q4 (6 pts) – Construire `tableau_final` (écart au premier)

On part de `tableau_temps` trié par temps total croissant : tuples `(dossard, nom, temps_total)`.

Objectif : construire `tableau_final`, liste de listes : - pour le premier :
`[dossard, nom, temps_premier]` - pour les suivants :
`[dossard, nom, temps_total - temps_premier]` (écart au premier)

```
tableau_final = []
difference_temps = 0
premier = True
for ligne in tableau_temps:
    coureur = [ligne[0]]
    coureur.append(ligne[1])
    if premier:
        temps_premier = ligne[2]
        coureur.append(temps_premier)
        premier = False
    else:
        difference_temps = ligne[2] - temps_premier
        coureur.append(difference_temps)
    tableau_final.append(coureur)
```

Barème Q4 (6 pts)

- 2 pts : mémorisation du temps du premier + ajout correct
- 2 pts : calcul d'écart correct `ligne[2] - temps_premier`
- 2 pts : construction et ajout corrects dans `tableau_final`

Partie B – SQL / Bases de données

Q5 (8 pts) – Pourquoi la clé primaire de `Temps` est `(numDossard, numEtape)` ?

Chaque ligne de `Temps` représente **le temps d'un coureur pour une étape**. - `numDossard` seul n'est pas unique : un coureur a **plusieurs** temps (un par étape). - `numEtape` seul n'est pas unique : une étape concerne **plusieurs** coureurs. Donc le couple `(numDossard, numEtape)` identifie **un enregistrement unique** : « ce coureur, à cette étape ».

Barème Q5 (8 pts)

- 4 pts : expliquer pourquoi `numDossard` seul ne suffit pas
- 4 pts : expliquer pourquoi `numEtape` seul ne suffit pas + conclusion sur l'unicité du couple

---

Q6 (6 pts) – Expliquer la requête

```
SELECT nomCoureur
FROM Coureurs
WHERE Equipe = 'Cofidis';
```

Elle renvoie **les noms des coureurs** dont l'équipe est **Cofidis**.

Barème Q6 (6 pts)

- 3 pts : « noms des coureurs »
- 3 pts : condition de filtrage sur l'équipe Cofidis

Q7 (6 pts) – Dates des étapes « contre-la-montre »

```
SELECT Date
FROM Etapes
WHERE Type = 'contre-la-montre';
```

Barème Q7 (6 pts)

- 3 pts : `SELECT`/`FROM` corrects
- 3 pts : `WHERE` correct sur `Type`

Q8 (10 pts) – Directeur sportif du coureur `BARDET Romain`

On relie l'équipe du coureur (`Coureurs.Equipe`) à la table `Equipes` (clé primaire `nomEquipe`) pour accéder à `directeurSportif`.

```
SELECT directeurSportif
FROM Equipes
JOIN Coureurs ON Coureurs.Equipe = Equipes.nomEquipe
WHERE nomCoureur = 'BARDET Romain';
```

Barème Q8 (10 pts)

- 4 pts : jointure correcte `Equipes` ↔ `Coureurs`
- 3 pts : sélection du bon attribut (`directeurSportif`)
- 3 pts : filtre correct sur `BARDET Romain`

Q9 (10 pts) – Pourquoi ces deux requêtes provoquent une erreur ?

```
INSERT INTO Temps VALUES (1, 5, 14267);
INSERT INTO Etapes VALUES (5, 'Montagne', 'Pau', 'Laruns', 05/07/2023);
```

Explication attendue (intégrité référentielle) :
`Temps.numEtape` est une **clé étrangère** qui doit référencer une étape existante dans `Etapes.numEtape`.
Si l'étape `5` n'existe pas encore dans `Etapes` au moment du premier `INSERT`, alors l'insertion dans `Temps` viole la contrainte → **erreur**.

> Remarque : selon le SGBD, le format de la date peut aussi poser problème (guillemets/format), mais l'essentiel en NSI est l'intégrité référentielle.

Barème Q9 (10 pts)

- 8 pts : mention claire « clé étrangère / étape 5 pas encore insérée dans `Etapes` »
- 2 pts : remarque sur le format/typage de date (si pertinent et bien formulé)

Q10 (4 pts) – Modification à effectuer

Il faut **insérer l'étape d'abord**, puis les temps qui la référencent :

1. `INSERT INTO Etapes ...`
2. `INSERT INTO Temps ...`

Barème Q10 (4 pts)

- 4 pts : inversion de l'ordre + justification par la contrainte de clé étrangère

---

Q11 (6 pts) – Temps total de course de `BARDET Romain`

On somme `tempsRealise` sur toutes les étapes du coureur.

```
SELECT SUM(tempsRealise)
FROM Temps
JOIN Coureurs ON Temps.numDossard = Coureurs.numDossard
WHERE nomCoureur = 'BARDET Romain';
```

Barème Q11 (6 pts)

- 2 pts : usage correct de `SUM(tempsRealise)`
- 2 pts : jointure correcte `Temps` ↔ `Coureurs` (ou sous-requête équivalente)
- 2 pts : filtre correct sur `BARDET Romain`

Corrigé détaillé – Exercice 3 (Terminale NSI décembre 2025)

Récursivité, structures de données, POO – Barème sur 60

Barème proposé (sur 60)

Question	Points
Q1	6 pts
Q2	6 pts
Q3	10 pts
Q4	3 pts
Q5	4 pts
Q6	8 pts
Q7	3 pts
Q8	4 pts
Q9	6 pts
Q10	4 pts
Q11	2 pts
Q12	4 pts
Total	**60 pts**

Partie A

Q1 (6 pts)

Pour n = 8 (bonbons 0 à 7), l'ordre des bonbons mangés est : 0, 3, 6, 2, 7, 5, 1
Le bonbon restant est : **4**.

Q2 (6 pts)

Une erreur **NameError** signifie que Python rencontre un nom non défini.

Ici `true` est incorrect : le booléen s'écrit `True`.

```
collier = [True for i in range(8)]
```

Q3 (10 pts)

```
def dernier(n):
    collier = [True for i in range(n)]
    indice = 0
    collier[indice] = False
    for etape in range(n-1):
        nb_bonbons_vus = 0
        while nb_bonbons_vus < 3:
            indice += 1
            if indice == n:
                indice = 0
            if collier[indice] == True:
                nb_bonbons_vus += 1
        collier[indice] = False
    return indice
```

Partie B

Q4 (3 pts)

Une file est une structure **FIFO** (First In First Out).

Q5 (4 pts)

Après les opérations, la file contient :

```
(Tête) 3 4 1 2 (Queue)
```

Q6 (8 pts)

```
def dernier_file(n):
    f = File()
    for i in range(n):
        f.enfile(i)
    f.defile()   # on mange le 0
    restants = n - 1
    while restants > 1:
        for _ in range(2):
            f.enfile(f.defile())
        f.defile()
        restants -= 1
    return f.defile()
```

Partie C

Q7 (3 pts)

`pred`, `valeur`, `succ` sont des **attributs**.

Q8 (4 pts)

a = 1
b = 2

Q9 (6 pts)

```
def creer_collier(n):
    premier = Bonbon(0)
    actuel = premier
    for i in range(1, n):
        nouveau = Bonbon(i)
        actuel.succ = nouveau
        nouveau.pred = actuel
        actuel = nouveau
    actuel.succ = premier
    premier.pred = actuel
    return premier
```

Q10 (4 pts)

Après suppression du bonbon 0, le collier est :

1 ↔ 2 ↔ 3 ↔ 1

Q11 (2 pts)

Bonne réponse : **C**

Q12 (4 pts)

```
def dernier_chaine(n):
    bonbon = creer_collier(n)
    while bonbon != bonbon.succ:
        bonbon.pred.succ = bonbon.succ
        bonbon.succ.pred = bonbon.pred
        bonbon = bonbon.succ.succ.succ
    return bonbon.valeur
```

Fin du corrigé.