# Bac blanc NSI — décembre 2025

Sujet

Bruno DARID — Décembre 2025

---

# 🎓 Bac Blanc NSI – LBO décembre 2025

**Durée : 3 heures 30**
**Calculatrice interdite**

---

## Exercice 1 Récursivité

> Barème : 6 points

Au rugby, une équipe peut marquer, pour simplifier :
• soit 3 points (pénalité) ;
• soit 5 points (essai non transformé) ;
• soit 7 points (essai transformé).

### Partie A

On souhaite savoir s'il est possible d'obtenir un score donné avec uniquement des pénalités, c'est-à-dire avec une succession de "coups" à 3 points.

1. Écrire une fonction `possible_avec_penalites_seules` qui prend en paramètre un score et qui renvoie `True` si le score passé en paramètre peut être marqué uniquement avec des pénalités.
   Exemple:

```
>>> possible_avec_penalites_seules(15)
True
>>> possible_avec_penalites_seules(10)
False
```

2. Recopier et compléter le tableau suivant qui précise, pour les scores de 0 à 10, les évolutions du score menant à un total donné et le nombre de façons différentes d'obtenir ce total. Par exemple, pour obtenir un score de 8, en partant de 0, il y a 2 possibilités :

- Soit l'équipe marque un essai non transformé, atteignant 5 points, puis une pénalité, atteignant 8 points ;
- Soit l'équipe marque une pénalité, atteignant 3 points, puis un essai non transformé, atteignant 8 points.

| Score | Liste des solutions | Nombre de solutions |
|---|---|---|
| 0 | [0] | 1 |
| 1 | [] | 0 |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | [0, 5, 8] ; [0, 3, 8] | 2 |
| 9 | | 1 |
| 10 | [0, 3, 10] | 3 |

En notant $f(n)$ le nombre de possibilités d'obtenir le score $n$ , on admet que pour $n > 6$, on a la relation suivante :

$$f(n) = f(n-3) + f(n-5) + f(n-7)$$

3. Vérifier cette relation pour $n = 10$ à l'aide du tableau établi à la question 2.

> On veut écrire une fonction récursive `nb_solutions` , qui prend en paramètre un entier positif quelconque correspondant à un score, et qui renvoie le nombre de façons
> d'obtenir ce score donné.

4. Déterminer tous les cas de base , pour chaque entier `n` de 0 à 6, de cette fonction récursive.
5. Écrire la fonction récursive `nb_solutions`, qui prend en paramètre un entier positif quelconque correspondant à un score, et qui renvoie le nombre de possibilités d'obtenir ce score donné.
6. Lors de l'appel de `nb_solutions(score)`, on se rend compte que le nombre d'appels récursif augmente très rapidement lorsque score augmente. Nommer
   une méthode algorithmique optimisant le nombre d'appels récursifs.

> On veut écrire une fonction récursive solutions_possibles , qui prend en paramètre un entier positif quelconque correspondant à un score, et qui renvoie la liste composée de toutes les listes représentant les possibilités d'obtenir ce score .

Par exemple :

```
>>> solutions_possibles(8)
[[0, 5, 8], [0, 3, 8]]
```

7. Déterminer quelles lignes du tableau permettent de construire rapidement la liste renvoyée par l'appel `solutions_possibles(11)`.
8. Recopier et compléter la fonction solutions_possibles suivante, qui prend en paramètre un score et qui renvoie la liste des possibilités d'obtenir ce score :

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

## Exercice 2 Programmation & BDD

> Barème : 8 points

Le Tour de France est une course cycliste qui se déroule chaque année. Chaque jour, les coureurs s'affrontent pour remporter l'étape du jour, ce qui détermine un classement d'étape. Le coureur avec le temps cumulé le plus bas sur l'ensemble des étapes mène le classement général. Chaque participant est repéré par un dossard et appartient à une équipe. En 2023, 22 équipes de 8 coureurs, soit 176 cyclistes ont pris le départ du tour.

**Partie A**

Dans cette partie, nous allons utiliser trois dictionnaires.
Le premier, appelé `participants`, a pour clés les noms complets des coureurs et pour valeurs les numéros de dossard correspondants. Le deuxième, appelé `temps_etapes`, utilise les numéros de dossard comme clés et contient une liste des temps d'arrivée de chaque étape en seconde. Le troisième, appelé `classement_general`, utilise également les numéros de dossard comme clés et indique le classement général mis à jour à la fin de chaque nouvelle étape.
Par exemple, à la fin de la quatrième étape, voilà les trois premiers éléments de ces dictionnaires :

```
participants = {"VINGEGAARD Jonas" : 1, "BENOOT Tiesj" : 2, "KELDERMAN Wilco": 3, ...}
temps_etapes = {1: [15781, 17199, 16995, 15928], 2: [15960, 17199, 16995, 15928], ...}
classement_general = {1: 6, 2: 30, 3: 13, ...}
```

1. En utilisant ces dictionnaires, écrire une instruction permettant d'obtenir :
    - le numéro de dossard de PHILIPSEN Jasper;
    - le classement général de PHILIPSEN Jasper;
    - le temps, en seconde, mis par le cycliste PINOT Thibaut pour courir la quatrième étape.
2. Écrire une fonction `calcul_temps_total` qui a pour paramètre le numéro d'un dossard `d` et qui renvoie le temps total en seconde mis par ce coureur depuis le départ du tour de France.
3. Le dictionnaire `temps_etapes` étant remis à jour après la fin d'une étape, recopier et compléter les lignes 8, 9 et 14 du programme suivant afin que le dictionnaire `classement_general` soit aussi mis à jour .

```
1  classement = []
2
3  for numero_dossard in temps_etapes:
4      element = (numero_dossard,
                  calcul_temps_total(numero_dossard))
5      classement.append(element)
6      pos = len(classement) - 2
7
8      while pos >= 0 and element[...] < classement[pos][...]:
9          classement[pos + 1] = ...
10         pos = pos - 1
11         classement[pos + 1] = element
12
13 for i in range(len(classement)):
14     classement_general[...] = i + 1
```

On suppose qu'on dispose d'un tableau `tableau_temps` composé de tuples contenant le numéro du dossard, le nom, et le temps total en seconde de chaque coureur, trié par ordre croissant de temps. On donne ci -dessous un aperçu du début du tableau :

```
tableau_temps = [(1 , "VINGEGAARD Jonas", 65903),
                         ( 3, "KELDERMAN Wilco", 65987),
                         ( 2, "BENOOT Tiesj", 66082),
                         ...]
```

On souhaite créer une variable Python `tableau_final` de type liste de listes :

```
[[1, "VINGEGAARD Jonas", 65903],
 [3, "KELDERMAN Wilco", 84],
 [2, "BENOOT Tiesj", 179]]
...
```

Pour le premier, la troisième valeur de la première liste est le temps total mis par le vainqueur. Pour les autres coureurs, la troisième valeur des autres listes est l 'écart de temps mis avec le premier.
Par exemple : 84 = 65987 - 65903 et 179 = 66082 - 65903.

4. Recopier et compléter le programme pour qu'il en soit ainsi:

```
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] - ...
        coureur.append(...)
    tableau_final.append(...)
```

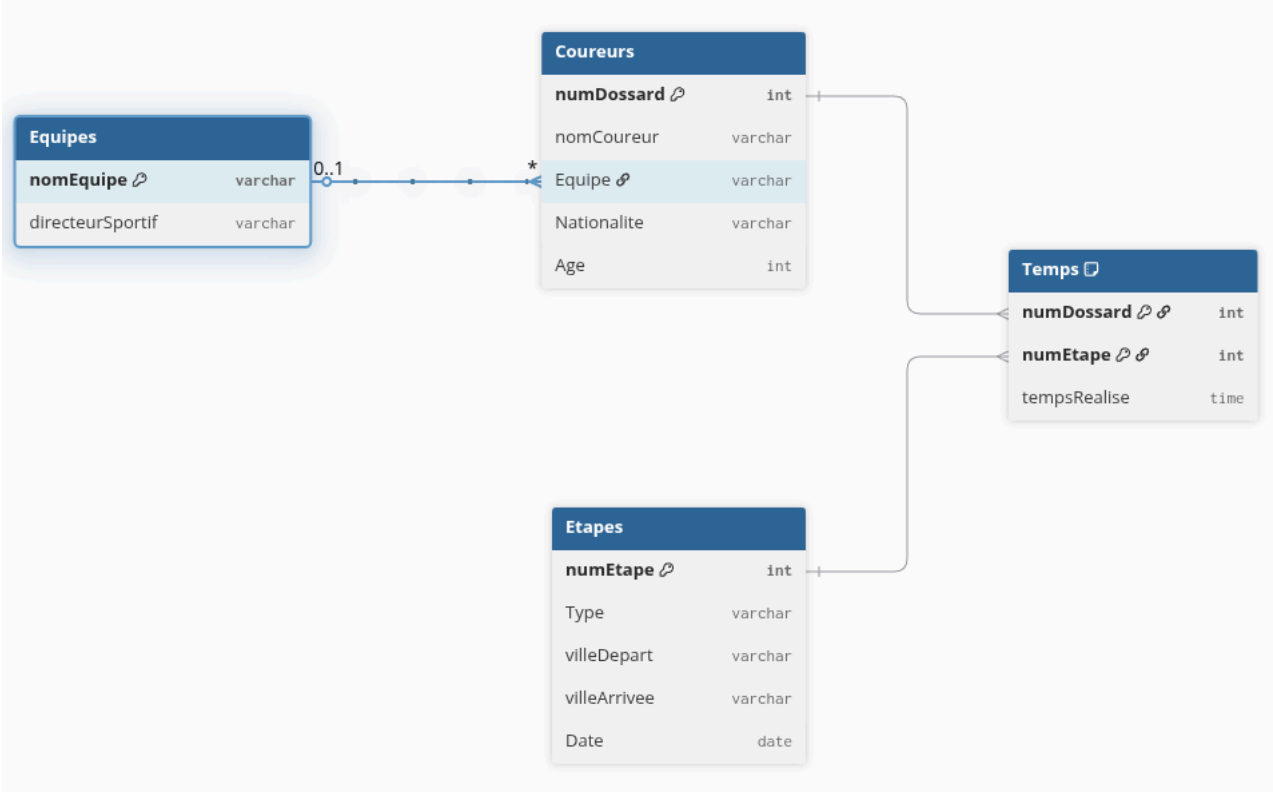
**Partie B**

Dans cet exercice, on pourra utiliser les mots clés suivants du langage SQL :

```
SELECT , FROM , WHERE , JOIN , ON, INSERT INTO , VALUES , MIN, MAX, OR, AND et ORDER BY
```

Si `propriete` est un des attributs d'une relation, les fonctions d'agrégation `MIN(propriete)` , `MAX(propriete)` , `SUM(propriete)` renvoient, respectivement, la plus petite, la plus grande valeur et la somme des valeurs des attributs sélectionnés.
On considère la base de données du tour de France 2023 dont le schéma relationnel est donné ci -dessous :

Figure 1. Schéma Relationnel




🚨 **Le sujet original comportait la phrase ci-après, qui perd son sens avec la schématisation choisie ici**

Dans ce schéma, les clés primaires sont soulignées et les clés étrangères sont précédées du symbole #.

5. Expliquer pourquoi, dans la relation `Temps`, il est nécessaire de prendre le couple `(numDossard , NumEtape)` comme clé primaire.
6. Expliquer ce que renvoie la requête SQL suivante :

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

7. Écrire une requête SQL permettant d'obtenir les dates de toutes les étapes de type `contre-la-montre` du tour de France 2023.
8. Écrire une requête SQL permettant d'obtenir le nom du directeur sportif du coureur BARDET Romain.
9. À la fin de la cinquième étape, on veut actualiser la table `Temps` avec les données du jour. Expliquer pourquoi la suite des deux requêtes SQL ci -dessous provoque une erreur .

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

10. Expliquer quelle modification est à effectuer pour apporter une solution au problème constaté à la question précédente.
11. Écrire une requête SQL donnant le temps total en course mis par BARDET Romain depuis le départ du tour de France 2023.

## Exercice 3 POO & SDD

Barème : 6 points

Cet exercice porte sur la gestion des bugs, l'algorithmique, les structures de données et la programmation orientée objet.

**Partie A**

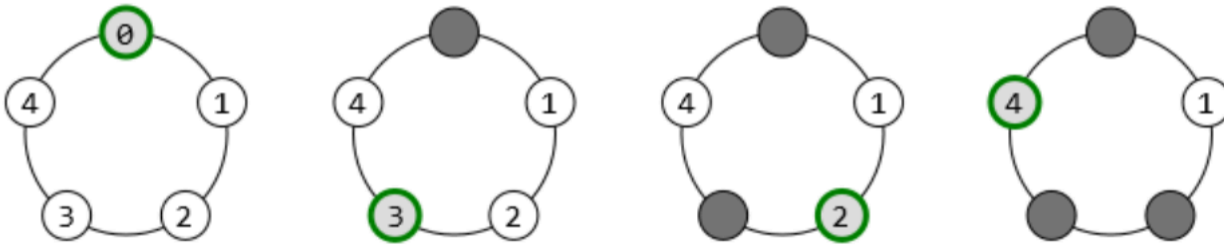
Un jour, Bob s'apprête à manger un collier de bonbons, et se pose la question suivante:

«*Si je mange un bonbon sur trois, encore et encore jusqu'à ce qu'il n'en reste qu'un seul, quel sera le dernier bonbon restant ?* »

fig1

*Figure 1. Collier de bonbons*

Pour un collier ayant 5 bonbons, il décide de les numéroter de 0 à 4. Il commence par manger le bonbon d'indice 0, se décale de trois bonbons et mange ensuite celui d'indice 3. En répétant la démarche, il mange ensuite le bonbon d'indice 2 et enfin celui d'indice 4.
Les indices des bonbons mangés sont donc, dans l'ordre, 0, 3, 2 et 4. Le bonbon restant est celui d'indice 1.




etapes

*Figure 2. Les étapes pour un collier de 5 bonbons*

1. Donner les indices dans l'ordre dans lequel les bonbons sont mangés dans le cas où le collier possède initialement 8 bonbons et l'indice du bonbon restant.

Afin de répondre à la question dans un cadre général, Bob décide de formaliser le problème. Il considère un collier de n bonbons numérotés de `0` à `n - 1`, où `n` est un entier strictement positif. Bob vient d'étudier en classe les valeurs booléennes. Il se dit qu'il peut représenter avec Python le collier par une liste `collier` telle que, pour toute valeur entière de `i` comprise entre `0` et `n - 1`, la valeur booléenne `collier[i]` vaut `True` si le bonbon d'indice `i` du collier est encore présent, et vaut `False` si le bonbon d'indice `i` du collier a été mangé.

Dès lors, il envisage l'algorithme suivant :

- on initialise une liste collier représentant n bonbons qui n'ont pas encore été mangés;
- on commence par manger le bonbon à l' indice 0;
- tant qu'il reste des bonbons à manger :
    - on détermine l' indice du prochain bonbon à manger dans la liste `collier`;
    - on mange le bonbon à cet indice;
- on renvoie l'indice du dernier bonbon mangé.

Afin de créer la liste collier décrite ci -dessus, Bob saisit dans la console l'instruction `collier = [true for i in range(8)]`. Il obtient alors le message d'erreur suivant :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module> NameError: name 'true' is not defined
```

2. Expliquer ce qu'est une erreur de type `NameError` et comment la corriger dans l'instruction proposée.

Bob écrit ensuite le code d' une fonction dernier qui prend en paramètre le nombre de bonbons n et renvoie l'indice du dernier bonbon restant. On fournit ci -après une partie du code de la fonction dernier.

```
 1 def dernier(n):
 2     collier = ...
 3     indice = 0
 4     collier[indice] = False
 5     for etape in range(n-1):
 6         nb_bonbons_vus = 0
 7         while nb_bonbons_vus ...:
 8             indice += 1
 9             if ...:
10                 indice = 0
11             if ...:
12                 nb_bonbons_vus += 1
13         collier[indice] = ...
14     return indice
```

3. Recopier et compléter les lignes 2, 7, 9, 11 et 13 du code de la fonction `dernier`.

**Partie B**

Bob se dit qu ’une structure de file lui permettrait de résoudre astucieusement le problème des bonbons.
On considère la classe `File` dont on fournit ci -après l’interface.

```
 1 class File:
 2    """Classe définissant une structure de file"""
 3
 4    def __init__(self):
 5        """Initialise une file vide"""
 6
 7    def est_vide(self):
 8        """Renvoie le booléen indiquant
 9         si la file est vide""" 10
11    def enfile(self, x):
12        """Place x à la queue de la file"""
13
14    def defile(self):
15        """Retire et renvoie l'élément placé à la
16        tête de la file
17        Provoque une erreur si la file est vide 18        """
19
20    def affiche(self):
21        """Affiche la file"""
```

Le code Python ci -après montre un exemple d’utilisation de la classe File.

```
>>> f = File()
>>> f.enfile(0)
>>> f.enfile(1)
>>> f.affiche()
(Tête) 0 1 (Queue)
```

L’acronyme LIFO signifie «Last In First Out» à savoir «Dernier entré, premier sorti». L’acronyme FIFO signifie «First In First Out» à savoir «Premier entré, premier sorti».

4. Donner l’acronyme le plus adapté à la structure de donnée File.
5. Déterminer l’affich age réalisé lors de l’exécution des instructions ci -après .

```
>>> f = File()
>>> for x in [0, 1, 2, 3, 4]:
>>>     f.enfile(x)
>>> f.defile()
>>> f.enfile(f.defile())
>>> f.enfile(f.defile())
>>> f.affiche()
```

6. Écrire le code de la fonction `dernier_file` qui prend en paramètre le nombre de bonbons `n` et renvoie l’indice du dernier bonbon restant.

**Partie C**

Bob souhaite utiliser la structure de données «liste doublement chaînée». Une telle liste est composée de maillons contenant chacun trois informations :

- une valeur ;
- un prédécesseur et un successeur qui sont tous deux des maillons.

Cette structure se prête bien au problème des bonbons : dans un collier, un bonbon est précédé et suivi par d’autres bonbons. Le successeur du dernier bonbon est le premier et le prédécesseur du premier, le dernier.

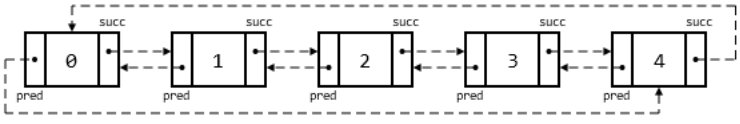



fig3

*Figure 3. Collier de cinq Bonbon*

Bob crée la classe Bonbon ci-après:

```
class Bonbon:
    def __init__(self, valeur):
        self.pred = None  # prédécesseur de ce bonbon
        self.valeur = valeur
        self.succ = None  # successeur de ce bonbon
```

7. Donner le terme correspondant aux variables `pred`, `valeur` et `succ` dans le vocabulaire de la programmation orientée objet.

Les instructions ci -dessous permettent de représenter un collier de trois bonbons de valeurs 0, 1 et 2.

```
>>> zero = Bonbon(0)
>>> un = Bonbon(1)
>>> deux = Bonbon(2)
>>> zero.succ = un
>>> un.pred = zero
>>> un.succ = deux
>>> deux.pred = un
>>> deux.succ = zero
>>> zero.pred = deux

>>> a = zero.succ.valeur
>>> b = un.succ.succ.pred.valeur
```

8. Déterminer les valeurs des variables `a` et `b` après l'exécution de ces instructions.

La fonction `creer_collier` prend en paramètre un entier `n` strictement positif représentant la taille d'un collier et renvoie un objet de type `Bonbon` représentant le premier bonbon (de valeur 0) du collier. On prendra soin de faire se succéder et précéder les différents bonbons ainsi que de « refermer » le collier en liant le dernier bonbon au premier.

9. Recopier et compléter les lignes 5, 6, 7, 9 et 10 du code de la fonction `creer_collier`, donné ci -après.

```
def creer_collier(n):
    premier = Bonbon(0)
    actuel = premier
    for i in range(1, n):
        nouveau = Bonbon(...)
        actuel.succ = ...
        ...
        actuel = nouveau  9    ...
    ...
    return premier
```

On considère le code Python suivant.

```
>>> bonbon = Bonbon(3)
>>> bonbon.pred = bonbon
>>> bonbon.succ = bonbon
```

À l'issue de l'exécution de ce code, on obtient la liste doublement chaînée représentée ci -dessous.

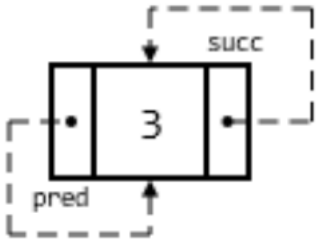



fig4

*Figure 4. Un collier d'un seul Bonbon*

10. On considère le code Python suivant.

```
>>> premier = creer_collier(4)
>>> premier.pred.succ = premier.succ
>>> premier.succ.pred = premier.pred
>>> bonbon = premier.succ
```

Dessiner une représentation du « collier » dont le premier élément est l'objet bonbon obtenu à l'issue de l'exécution du code Python ci -dessus.

11. Dans le cas où il ne reste qu'un bonbon, donner l'expression qui s'évalue à `True`, parmi les quatre propositions ci -dessous:

- Proposition A : `valeur.succ == valeur.bonbon`
- Proposition B : `pred == succ`
- Proposition C : `bonbon.valeur == bonbon.succ.valeur`
- Proposition D : `bonbon.valeur == succ.valeur`

12. Recopier et compléter les lignes 3, 4, 5 et 6 du code de la fonction `dernier_chaine`, donné ci -après, qui prend en paramètre le nombre de bonbons `n` et renvoie la valeur du dernier bonbon restant .

```
def dernier_chaine(n):
    bonbon = creer_collier(n)
    while ... != ...:
        bonbon.pred.succ = ...
        ... = bonbon.pred
        bonbon = ... # décalage de 3 bonbons
    return bonbon.valeur
```