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 2

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 17 pages numérotées de 1/17 à 17/17.

Le sujet est composé de trois exercices indépendants.

Le candidat traite les trois exercices.

Exercice 1 (6 points)

Cet exercice porte sur l'architecture matérielle, les systèmes d'exploitation, et les structures de données linéaires en Python.

Un système d'exploitation permet d'exécuter plusieurs applications à la fois en donnant l'impression qu'elles fonctionnent simultanément. En réalité, le système répartit le temps de calcul du processeur entre les différents processus, de sorte qu'ils s'exécutent chacun à leur tour très rapidement.

Chaque application peut être représentée par un ou plusieurs processus, gérés par le système d'exploitation. Même si un seul processus utilise réellement le processeur à un instant donné, cette alternance rapide donne l'impression que tout s'exécute en même temps. Lorsque le système interrompt un processus pour en exécuter un autre, on parle de *préemption*.

Partie A

1. Recopier et compléter le schéma ci-dessous avec les termes suivants :« élu », « prêt », « bloqué », « élection », « blocage », « déblocage ».

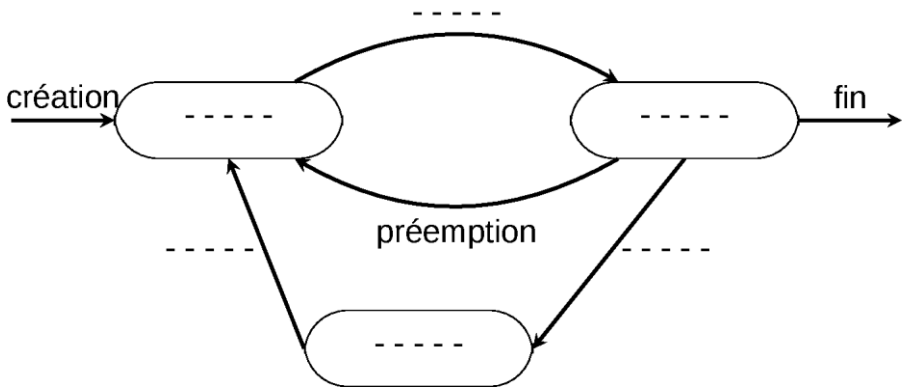


Figure 1. États d'un processus.

On peut imaginer, par exemple, une application de streaming musical qui se compose de trois processus :

- le processus P1 gère l'interface utilisateur et interagit avec les composants visibles : listes de morceaux, boutons de lecture-pause, etc. ;
- le processus P2 assure le téléchargement de la musique et l'écrit dans une mémoire cache locale ;
- le troisième processus P3, qu'on pourrait appeler *lecteur audio*, assure le décodage audio et envoie le flux de données au système pour lecture.

Voici une situation de fonctionnement :

- l'utilisateur décide de mettre l'application en arrière-plan, cachant ainsi l'interface utilisateur ;

- le processus P2 attend un accès à la carte WIFI pour recharger des données ;
- le processus P3 décode de la musique et l'envoie au système.

2. Indiquer l'état de chacun des trois processus P1, P2 et P3 dans cette situation.

3. Expliquer, de façon générale, quand est-ce qu'un interblocage de processus peut survenir.

On considère plusieurs ressources dont un processeur graphique (GPU), un microphone (MIC), une caméra (CAM) et un processeur dédié au calcul (CAL). On suppose que chacune de ces ressources ne peut être utilisée que par un seul processus à la fois.

On considère de plus quatre processus nommés P1, P2, P3 et P4. Le tableau suivant récapitule les ressources utilisées par chacun des quatre processus, dans l'ordre où chacun des processus les demande :

P1	P2	P3	P4
demander MIC demander CAL libérer CAL libérer MIC demander CAM demander GPU libérer CAM libérer GPU	demander CAL demander MIC libérer MIC libérer CAL demander CAM libérer CAM	demander CAM libérer CAM demander CAL demander MIC libérer MIC libérer CAL demander GPU libérer GPU	demander GPU demander CAL demander CAM libérer CAM libérer CAL demander MIC libérer GPU libérer MIC

4. Les processus s'exécutent de manière concurrente. Justifier qu'une situation d'interblocage peut se produire.

5. Expliquer l'intérêt d'utiliser une machine équipée de plusieurs processeurs plutôt qu'une machine équipée d'un seul processeur.

6. Décrire un avantage et un inconvénient des systèmes sur puces, tels que ceux utilisés dans les smartphones.

Partie B

On s'intéresse à un ordonnanceur de type tourniquet (round-robin), dans lequel une durée appelée *quantum* est fixée à 2 ms. Chaque processus, lorsqu'il est défilé de la file d'attente, s'exécute sur le processeur pour une durée au plus égale au *quantum* :

- s'il termine son exécution avant ou à la fin du quantum, un autre processus est défilé de la file d'attente, et celui-ci bénéficie d'un nouveau quantum pour s'exécuter.
- s'il n'a pas terminé son exécution, il est mis en queue de la file d'attente.

De plus, d'autres processus peuvent être ajoutés à la file d'attente au fur et à mesure qu'ils arrivent.

On considère quatre processus nommés P1, P2, P3 et P4. Le tableau suivant donne les informations temporelles les concernant.

Processus	Instant d'arrivée	Temps d'exécution total
P1	0 ms	6 ms
P2	1 ms	4 ms
P3	3 ms	5 ms
P4	5 ms	3 ms

7. Recopier et compléter le chronogramme suivant en indiquant quel processus utilise le processeur à chaque instant, de 0 ms à la fin de l'exécution de tous les processus. À titre d'exemple, on a déjà indiqué sur le chronogramme que le processus P1 s'exécute entre les instants 0 ms et 2 ms. De plus on a indiqué sous l'axe du temps les instants d'arrivée des quatre processus.

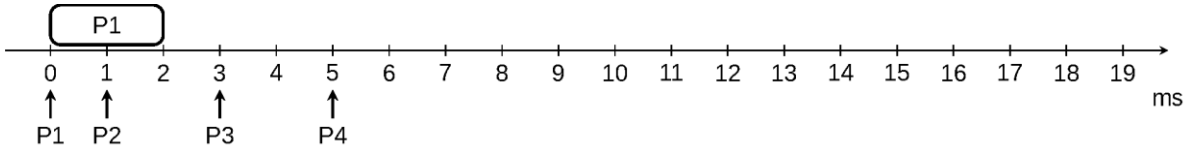


Figure 2. Chronogramme à compléter.

On souhaite modéliser en Python le comportement de l'algorithme du tourniquet. Chaque processus est représenté par un dictionnaire contenant les clés `'nom'`, `'arrivee'` et `'temps'`, ayant pour valeurs correspondantes le nom (de type `string`), l'instant d'arrivée (de type `int`) et le temps d'exécution restant en millisecondes (de type `int`).

On définit donc ci-dessous quatre variables `P1`, `P2`, `P3` et `P4` représentant les quatre processus considérés ci-avant.

```
1 P1 = {'nom': 'P1', 'arrivee': 0, 'temps': 6}
2 P2 = {'nom': 'P2', 'arrivee': 1, 'temps': 4}
3 P3 = {'nom': 'P3', 'arrivee': 3, 'temps': 5}
4 P4 = {'nom': 'P4', 'arrivee': 5, 'temps': 3}
```

Le quantum est fixé à 2 ms grâce à une variable `quantum` (de type `int`).

De plus, on suppose qu'on dispose d'une implémentation de file en Python à travers les fonctions suivantes :

- `creer_file_vide` qui ne prend aucun paramètre et qui renvoie une file vide ;

- `est_vide` qui prend en paramètre une file et qui renvoie un booléen indiquant si cette file est vide ;
- `enfiler` qui prend en paramètres une file et un élément et qui modifie la file en y ajoutant cet élément en queue (la valeur de retour est `None`) ;
- `defiler` qui prend en paramètre une file, qui modifie cette file en enlevant son élément en tête et qui renvoie cet élément.

8. Donner les instructions qui permettent de créer une file `fp` contenant les processus `P1`, `P2`, `P3` et `P4`, classés par ordre d'arrivée, le premier arrivé étant en tête de la file.

On souhaite créer une fonction `execute_un_processus` qui réalise une étape de l'algorithme du tourniquet. Plus précisément cette fonction prend en paramètres une file de processus non vide `file_d_attente` et un instant `t` donnant le début de cette étape. Elle extrait de la file le processus à exécuter, puis le remet si besoin dans la file d'attente avec le temps d'exécution restant. De plus cette fonction renvoie l'instant auquel cette étape se termine, qui est soit la fin de l'exécution du processus, soit la fin du quantum.

9. Recopier et compléter le code de la fonction `execute_un_processus` ci-dessous.

```
1 def execute_un_processus (file_d_attente, t):
2     processus = defiler(file_d_attente)
3     if processus['temps'] ... quantum:
4         processus['temps'] = ...
5         ...
6         return t + ...
7     else:
8         return t + ...
```

On se place maintenant dans le cas où tous les processus sont déjà arrivés, autrement dit tous les processus à traiter sont dans la file que l'on prend en entrée. On ne tiendra donc pas compte des temps d'arrivée dans la suite. On souhaite créer une fonction `execute_tous_processus` qui prend en paramètre une file de processus, qui simule le comportement de l'algorithme du tourniquet sur ces processus jusqu'à les avoir tous complètement exécutés, et qui renvoie à quel instant se termine la dernière de ces exécutions. On suppose que l'exécution de ces processus commence à partir de l'instant 0.

10. Recopier et compléter le code de la fonction `execute_tous_processus` ci-dessous.

```
1 def execute_tous_processus (file_d_attente):
2     t = 0
3     while ...
4         t = ...
5     return t
```

Exercice 2 (6 points)

Cet exercice porte sur la programmation Python en général, la programmation orientée objet en particulier et la structure de données d'arbre.

La WTA (Women's Tennis Association) est l'instance dirigeante du tennis professionnel international féminin, responsable de l'organisation du circuit « WTA Tour » et de l'établissement des classements mondiaux des joueuses. Dans cet exercice, on ne considère que des matchs de tennis en simple, c'est-à-dire des matchs qui opposent uniquement deux joueuses.

Partie A

On dispose d'une classe `Joueuse` pour modéliser une joueuse de tennis professionnelle du circuit WTA.

```
 1 class Joueuse:
 2     def __init__(self, nom, prenom, pays, age, point):
 3         ''' nom, prenom et pays sont de type str,
 4         age et point de type int '''
 5         self.nom = nom
 6         self.prenom = prenom
 7         self.pays = pays
 8         self.age = age
 9         # nombre de points WTA
10         self.point = point
11         # nombre de victoires
12         self.victoire = 0
13         # nombre de défaites
14         self.defaite = 0
```

1. Instancier l'objet `pegula` de la classe `Joueuse` qui modélise la joueuse Pegula Jessica de nationalité américaine (`"USA"`), âgée de 29 ans et ayant 6101 points WTA.

On considère les objets déjà instanciés `gauff`, `paloni`, `sabalenka` et `swiatek` de la classe `Joueuse` représentant respectivement les joueuses :

- Gauff Coco, américaine, âgée de 20 ans et ayant 6063 points WTA ;
- Paloni Marta, espagnole, âgée de 23 ans et ayant 4843 points WTA ;
- Sabalenka Aryna, biélorusse, âgée de 25 ans et ayant 10541 points WTA ;
- Swiatek Iga, polonaise, âgée de 22 ans et ayant 7470 points WTA.

2. Recopier et compléter le code ci-après de la méthode `ajouter_victoire` de la classe `Joueuse` permettant d'ajouter une victoire à la joueuse en question ainsi qu'une défaite à son adversaire objet de la classe `Joueuse`.

```
    def ajouter_victoire(self, adversaire):
        ...
        ...
```

Lors de la finale du dernier tournoi des masters Marta Paloni a été battue par Iga Swiatek.

3. Écrire une instruction utilisant la méthode `ajouter_victoire` pour prendre en compte l'issue de ce match.

On souhaite classer les joueuses à l'aide de leurs points WTA. Pour cela il est possible en Python redéfinir l'opérateur de comparaison $<$ (strictement inférieur) pour les objets de la classe `Joueuse` en utilisant les points WTA. Ainsi on obtient par exemple :

```
>>> swiatek < paloni
False
>>> swiatek < sabalenka
True
```

On utilise une liste Python pour stocker des objets de la classe `Joueuse`,

```
liste_joueuses = [swiatek, gauff, paloni, sabalenka, pegula]
```

que l'on souhaite trier dans l'ordre croissant des points WTA. Pour cela on choisit le tri par insertion dont le code est donné ci-dessous :

```
def tri_insertion(liste):
    for i in range(1, len(liste)):
        j = i
        while j > 0 and liste[j] < liste[j - 1]:
            # échange  les valeurs liste[j] et liste[j-1]
            liste[j], liste[j - 1] = liste[j - 1], liste[j]
            j = j - 1
```

4. Préciser, sans justifier, le coût temporel du tri par insertion d'une liste de n éléments dans le pire des cas.

On détaille les étapes du tri par insertion appliqué à la liste `liste_joueuses` dans le tableau ci-dessous, où chaque ligne correspond à un échange entre deux éléments de la liste.

Étape	Contenu de `liste_joueuses`
0	`[swiatek, gauff, paloni, sabalenka, pegula]`
1	`[gauff, swiatek, paloni, sabalenka, pegula]`
…	……

5. Recopier et compléter le tableau ci-dessus en ajoutant autant de lignes que nécessaires.

Un match de tennis se déroule en plusieurs *sets*, et chaque set en plusieurs *jeux*. Ayant fixé qui est la joueuse 1 et qui est la joueuse 2, on peut décrire l'issue d'un set à l'aide d'un couple d'entier : le premier entier donne le nombre de jeux gagnés par la joueuse 1 lors de ce set, le second le nombre de jeux gagnés par la joueuse 2 lors de ce set. Le score d'un match est alors décrit par une liste de tuples.

Par exemple, en mai 2024, Swiatek affronte Sabalenka en finale du tournoi « WTA 1000 » de Madrid. Ce match s'est déroulé en 3 sets : 7-5, 4-6, 7-6. Swiatek remporte donc la finale 2 sets à 1. La liste de tuples représentant le score de ce match est `[(7,5), (4,6), (7,6)]`.

Aucun cas d'égalité n'est possible.

Pour automatiser la gestion des tournois on crée une classe `Match`.

```
1 class Match:
2     def __init__(self, intitule, joueuse1, joueuse2):
3         self.intitule = intitule
4         self.joueuse1 = joueuse1
5         self.joueuse2 = joueuse2
6         self.gagnante = None
7         self.perdante = None
8         self.score = None
```

Cette classe est constituée de :

- `self.intitule`, un intitulé du match sous la forme d'une chaîne de caractères ;
- `self.joueuse1` et `self.joueuse2`, représentant les deux joueuses, deux objets de la classe `Joueuse` ;
- `self.gagnante` et `self.perdante`, représentant la joueuse victorieuse respectivement la joueuse perdante deux objets de la classe `Joueuse` ;
- `self.score`, représentant le score du match sous la forme d'une liste de tuples d'entiers.

Au terme d'un match une fois que le score est connu, on souhaite pouvoir le saisir à l'aide de la méthode `resultat_match(self, score)` de la classe `Match`. Cette méthode prend en paramètre un score sous la forme de liste de tuple de deux entiers et

- enregistre le score du match ;
- détermine la gagnante et la perdante en comptant le nombre de set(s) gagné(s) par chaque joueuse ;

- ajoute une victoire à la joueuse gagnante et une défaite à la joueuse perdante.

Par exemple, pour créer un objet `finale_mad_24` représentant le match Swiatek vs Sabalenka décrit précédemment, on utilise les deux instructions suivantes.

```
# instanciation de la finale Madrid 2024
finale_mad_24 = Match('finale Madrid 24', swiatek,
                      sabalenka)
# mise-à-jour du score du match
finale_mad_24.resultat_match([(7,5), (4,6), (7,6)])
```

6. Recopier et compléter le code de la méthode `resultat_match` de la classe `Match`. On ajoutera autant de lignes que nécessaire.

```
1     def resultat_match(self, score):
2         self.score = score
3         nb_set_joueuse1 = 0
4         nb_set_joueuse2 = 0
5         # code incomplet
```

Partie B

Afin de pouvoir modéliser chaque tour d'un tournoi de tennis à partir des quarts de finale uniquement, on décide d'utiliser un arbre binaire.

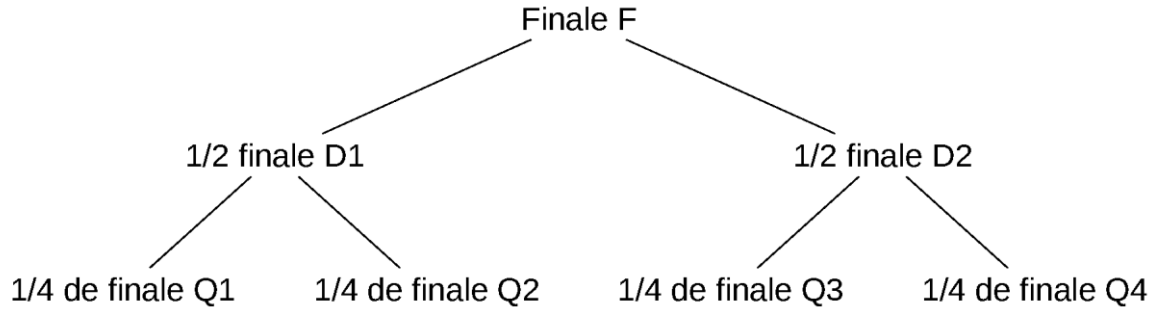


Figure 1. Tournoi de tennis.

7. Justifier qu'un tournoi de tennis peut effectivement être modélisé à l'aide d'un arbre binaire.

On décide d'implémenter un arbre binaire à l'aide de la classe `Arbre` ci-dessous :

```
1 class Arbre:
2     def __init__(self, racine, gauche, droit):
3         self.racine = racine
4         self.gauche = gauche
5         self.droit = droit
```

où `self.racine` est un objet de la classe `Match`, `self.gauche` et `self.droit`, des objets de la classe `Arbre`.

Dans le cas du tournoi de Madrid 2025 on connaît les matchs du tournoi à partir des quarts de finale, où `pilar`, `inie`, `jabeur` sont des objets de la classe `Joueuse` :

```
# quarts de finale
Q1 = Match("Quart de finale 1", gauff, pilar)
Q2 = Match("Quart de finale 2", paloni, inie)
Q3 = Match("Quart de finale 3", pegula, sabalenka)
Q4 = Match("Quart de finale 4", swiatek, jabeur)
# demi-finale
D1 = Match("Demi-finale 1", None, None)
D2 = Match("Demi-finale 2", None, None)
# finale
F = Match("Finale", None, None)
```

- La première demi-finale `D1` verra s'affronter les joueuses victorieuses des quarts de finales `Q1` et `Q2`.
- La seconde demi-finale `D2` verra s'affronter les joueuses victorieuses des quarts de finales `Q3` et `Q4`.
- La finale `F` verra s'affronter les joueuses victorieuses des demi-finales `D1` et `D2`.

8. Instancier la variable `tournoi` de la classe `Arbre` représentant ce tournoi depuis les quarts de finale en passant par les demi-finales jusqu'à la finale.

Dès qu'un match est joué, et donc dès que la gagnante est connue, on souhaite mettre à jour le match du tour suivant. Par exemple le premier quart de finale `Q1` s'est terminé sur le score de 6-4, 7-5, en faveur de la joueuse `gauff`. Elle participera donc en tant que `joueuse1` au premier match des demi-finales `D1`.

```
Q1.resultat_match([(6, 4), (7, 5)])
```

9. Compléter l'instruction suivante afin de mettre à jour dans `tournoi` la première joueuse de la première demi-finale comme étant `gauff`.

```
tournoi. ... = gauff
```

Cette façon de mettre à jour le tournoi n'est pas satisfaisante, on souhaite automatiser cette tâche à l'aide d'une méthode `mise_a_jour` de la classe `Arbre` qui parcourt l'arbre et dès que l'on rencontre un match dont la gagnante est connue, on la fait passer au tour suivant si ce n'est pas déjà fait. Cette méthode doit être récursive.

10. Rappeler ce qu'est un programme récursif.

11. Recopier et compléter les lignes 6, 7, 13, 14 et 16 du code ci-dessous de la méthode `mise_a_jour` permettant de compléter automatiquement les matchs du tournoi en les mettant à jour à partir des résultats des matchs des tours précédents.

```
1     def mise_a_jour(self):
2         """Met à jour les matchs de l'arbre"""
3         if self.racine.joueuse1 is None:
4             if self.gauche is not None:
5                 # mise à jour si gagnante à gauche
6                 if ...
7                     ... = self.gauche.racine.gagnante
8                 else:
9                     self.gauche.mise_a_jour()
10        if self.racine.joueuse2 is None:
11            if self.droit is not None:
12                # mise à jour si gagnante à droite
13                 if ...
14                     ... = self.droit.racine.gagnante
15                 else:
16                     ...
```

Exercice 3 (8 points)

Cet exercice porte sur les bases de données relationnelles, le langage SQL et la programmation Python, en particulier les dictionnaires.

Un club d'athlétisme organise une compétition de course à pied sur route. Deux épreuves sont proposées : une course de 5 km et une course de 10 km.

Partie A : Bases de données

Dans cet exercice, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY` ;
- réaliser des agrégations à l'aide de `COUNT`.

Le club souhaite gérer les inscriptions et les résultats de cette compétition à l'aide d'une base de données informatique constituée de deux tables : `coureur` et `epreuve`. Le schéma relationnel de cette base de données est représenté ci-dessous. Sur ce schéma, les clés primaires de chacune des tables sont soulignées et les clés étrangères sont précédées du symbole #.

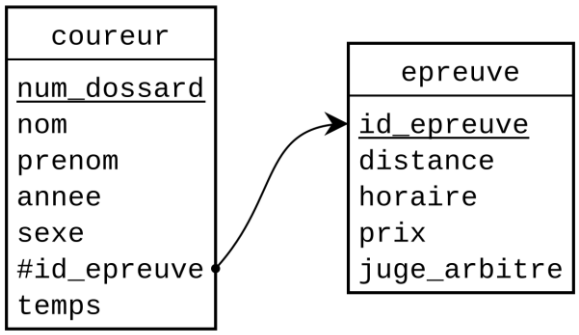


Figure 1. Schéma de la base de données

Chaque coureur ne peut participer qu'à une seule épreuve.

Relation `coureur`

L'identifiant du coureur est son numéro de dossard. Ce numéro est automatiquement incrémenté d'une unité à chaque nouvel enregistrement. Les autres champs doivent être saisis par l'organisateur lorsqu'il reçoit les bulletins d'inscription des coureurs. La codification du sexe est « H » pour les hommes et « F » pour les femmes. Les temps sont exprimés en secondes et sont initialisés avec la valeur 0.

Extrait de table `coureur`						
num_dossard	nom	prenom	annee	sexe	id_epreuve	temps
1	DA SILVA	José	1980	H	1	0
2	HANG LI	Léo	2005	H	2	0
3	BODIANE	Lola	2000	F	2	0
4	BRELET	Sandra	1972	F	1	0

Relation `epreuve`

Cette relation contient actuellement deux enregistrements, l'organisateur prévoyant d'ajouter de nouvelles distances dans le futur. La distance est exprimée en km et le prix de l'inscription en euros. Le champ `horaire` donne l'heure de départ de la course.

epreuve				
id_epreuve	distance	horaire	prix	juge_arbitre
1	5	10	10	GAVEAU Philippe
2	10	11	20	BOULA Awa

1. Expliquer le choix de `num_dossard` comme clé primaire pour la relation `coureur`.

2. Décrire ce que donne la requête SQL suivante :

```
SELECT nom, prenom
FROM coureur
ORDER BY nom ;
```

3. Écrire une requête SQL permettant d'établir le nom et prénom de toutes les femmes inscrites à la compétition.

4. Écrire une requête SQL permettant de connaître le nombre total d'inscrits à la compétition.

L'organisateur reçoit le bulletin d'inscription contenant les informations suivantes :

> Nom : REMY ; Prénom : Patrice
>
> Civilité : homme
>
> Course : 5 km
>
> Année de naissance : 1973

5. Écrire une requête SQL permettant d'insérer ce participant dans la base.

Le coureur dont le numéro de dossard est le 137 s'est blessé quelques jours avant l'épreuve, il ne pourra pas participer à la course.

6. Écrire une requête SQL qui permettra à l'organisateur de supprimer son inscription de la base de données.

Le coureur dont le numéro de dossard est le 256 a oublié sur quelle épreuve (5 km ou 10 km) il s'est inscrit. Il contacte l'organisateur pour qu'il lui rappelle la distance qu'il devra parcourir ainsi que l'horaire de son départ.

7. Écrire une requête SQL permettant de lui fournir ces informations.

Lorsque les coureurs franchissent la ligne d'arrivée, le juge arbitre note leur numéro de dossard et leur temps de course (en secondes) sur une feuille de pointage. Voici un extrait d'une telle feuille :

dossard	temps
57	1242
72	1845
183	1284
2	1285

Les temps de tous les coureurs ayant été enregistrés, on souhaite afficher le classement général de la catégorie *Master Femme* pour la course de 10 km. Cette catégorie regroupe les coureuses nées avant 1986.

8. Écrire une requête SQL renvoyant le numéro de dossard, le nom, le prénom et le temps de course de ces participantes classées par temps de course croissant.

Partie B : Programmation Python

Émilie, fille de l'organisateur et élève en classe de terminale spécialité NSI propose de réaliser une étude pluriannuelle des performances accomplies.

Pour enregistrer les informations elle crée pour chaque type de course un *dictionnaire de performances*. Les clés d'un tel dictionnaire sont les années où ce type de course a eu lieu. La valeur associée à une année est la liste des meilleurs temps réalisés dans chacune des six catégories suivantes (dans cet ordre).

- Junior homme (JH) et Junior femme (JF) : moins de 18 ans ;
- Sénior homme (SH) et Sénior femme (SF) : de 18 à 40 ans ;
- Master homme (MH) et Master femme (MF) : plus de 40 ans ;

Par exemple, le dictionnaire ci-dessous enregistre les performances pour les courses de 5 km. Il indique par exemple qu'en 2024 le meilleur temps réalisé dans la catégorie junior femme (JF) est de 1010 s, et celui dans la catégorie master homme (MH) de 1022 s.

```
dict_perf_5km = {2022 : [1020,1050, 900,1000,1018,1040],
                 2023 : [1010,1048,1100,1024,1080,1108],
                 2024 : [1012,1010,1000,1036,1022,1098],
                 2025 : [ 998,1028,1000, 959,1002, 980]}
```

9. Donner la valeur de l'expression `dict_perf_5km[2025][2]`

En 2026, pour la course de 5km, les meilleurs temps réalisés dans chaque catégorie sont les suivants :

Résultats par catégories - 2026					
JH	JF	SH	SF	MH	MF
1004	1016	1000	1140	1023	1024

10. Écrire l'instruction permettant d'ajouter ces informations dans le dictionnaire `dict_perf_5km`.

11. Écrire le code de la fonction `scratch` qui prend en paramètres un dictionnaire de performances `dico` et une année `annee`, et qui renvoie le meilleur temps (toutes catégories confondues) de cette année. On suppose que `annee` est une clé présente dans `dico`.
    On n'utilisera pas les fonctions natives `min` et `max` de Python.
    Par exemple, l'appel `scratch(dict_perf_5km, 2024)` renvoie `1000`.

Émilie propose maintenant la fonction suivante :

```
 1 def mystere(dico, cat):
 2     categories = ['JH','JF','SH','SF','MH','MF']
 3     s = 0
 4     nb = 0
 5     for i in range(len(categories)):
 6         if cat == categories[i]:
 7             i_cat = i
 8     for annee in dico.keys():
 9         s = s + dico[annee][i_cat]
10         nb = nb + 1
11     return s/nb
```

12. Indiquer la valeur affectée à la variable `i_cat` au cours de l'appel `mystere(dict_perf_5km, 'SH')`.

Un appel du type `mystere(dict_perf_5km,'SG')` pose problème car `'SG'` n'est pas une catégorie répertoriée.

13. Indiquer quel sera le message d'erreur renvoyé à la console parmi les choix suivants :
    - "`expected an indented block`";
    - "`takes 2 positionnals arguments but 3 were given`";
    - "`local variable 'i_cat' referenced before assignement`";
    - "`list index out of range`".
14. Proposer une assertion à ajouter entre la ligne 2 et la ligne 3 pour indiquer une éventuelle faute de saisie par un message à l'utilisateur du script.
15. Indiquer le résultat de l'appel `mystere(dict_perf_5km,'SH')`.

Émilie souhaiterait disposer d'une fonction `records` prenant en paramètre un dictionnaire de performances et renvoyant la liste des meilleurs temps enregistrés pour chaque catégorie et toutes années confondues. Par exemple, l'appel `records(dict_perf_5km)` devrait renvoyer `[998, 1010, 900, 959, 1002, 980]`. On suppose que les temps enregistrés pour les différentes catégories et les différentes années n'excèdent jamais 24 h.

16. Écrire le code de la fonction `records`.