Balthazar Patiachvili <balthazar.patiachvili@crans.org>

Composition 2024

Partie 1. Robot de dépôt de solutions chimiques

Question 1.1

On note A et B les deux exécutions parallèles de la fonction `déposer`. L'exécution des lignes suivantes pose un problème : `1A`, `2A`, `3A`, `1B`, `2B`, `3B`, `4A`, `4B`, … En effet, dans cette situation, l'exécution B de la fonction déplacera le bras au-dessus de sa propre éprouvette avant que l'exécution A n'ait déposé sa solution. Deux threads exécutant la fonction `déposer` sont alors susceptibles de déposer leur solution dans la même éprouvette.

Question 1.2

Cette solution ne règle pas le problème évoqué à la **Question 1.1** : si lors de deux exécutions parallèles de la nouvelle fonction `déposer`, les lignes suivantes sont exécutées dans cet ordre : `4A`, `5A`, `6A`, `4B`, `5B`, `6B`, alors la fonction `libre` peut renvoyer pour les deux fonctions la même valeur car la solution n'a pas encore été injectée dans l'éprouvette. Cette solution n'est donc pas satisfaisante.

Question 1.3

Cette version ne fonctionne pas non plus car le `return` de la ligne 8 sera exécuté avant le relâchement du mutex de la ligne 9 : le mutex ne sera donc jamais relâché et la fonction `déposer` ne pourra être exécutée qu'une unique fois.

On peut alors la corriger de la manière suivante :

```c
int déposer(struct formule *f) {
  pthread_mutex_lock(&mutex);
  int i = libre();
  aller(i);
  injecter(f);
  // Inversion des lignes 8 et 9 du programme initial
  pthread_mutex_unlock(&mutex);
  return i;
}
```

Question 1.4

On peut gérer les appels parallèles de la manière suivante :

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int déposer(struct formule *f) {
  struct codage code = encoder(f, "bras modèle 1");

  pthread_mutex_lock(&mutex);
  int i = libre();
  aller(i);
  injecter(&code);
  pthread_mutex_unlock(&mutex);
```

```
11    return i;
12  }
```

On met alors l'appel à la fonction `encoder` en dehors de la zone critique contrôlée par le mutex pour s'assurer qu'elle puisse bien s'exécuter en parallèle sur plusieurs threads, tout en s'assurant de la bonne utilisation des mutex comme pour la **Question 1.3**.

Question 1.5

Dans le cas de deux threads, il n'est pas possible d'assurer un temps d'attente égal à une heure à la seconde près car les fonctions `aller` et `analyser` sont bloquées par un mutex empêchant les deux threads de s'y rendre de manière concurrente. En effet : si la fonction `analyser` du premier thread qui appelle la fonction `analyser` met plus d'une heure à exécuter la fonction `analyser`, alors il sera impossible que le second thread exécute la fonction `analyser` exactement une heure après l'appel à la fonction `injection`, peu importe l'ordre d'exécution des instructions précédentes.

Question 1.6

Dans le cas d'un seul thread avec une précision de la seconde, alors il est possible d'assurer un temps d'attente exactement égal à une heure entre les appels à `injecter` et à `analyser`. En effet, le bras ne doit pas se déplacer, et on peut considérer que toutes les opérations et calculs effectuées après l'appel à `injecter` s'effectuent en moins d'une heure, et sont déterministes. En particulier, le temps d'exécution des appels à `injecter` ne devraient pas changer d'un appel à l'autre : on peut donc tester pour avoir une référence. En faisant un appel à la fonction `sleep()` de la bonne durée, égale à une heure moins le temps d'exécution de l'appel à `injecter` et du temps de déplacement à la bonne éprouvette (également déterministe) avec la fonction `aller`, on peut donc attendre exactement une heure entre les appels à `injecter` et à `analyser`.

Cependant, il n'est pas possible d'être précis à un cycle processeur près. En effet, l'ordonnanceur d'un processeur moderne n'exécutant pas forcément les instructions dans l'ordre séquentiel, il n'est pas possible, même dans le cas le plus basique d'un processeur à un seul cœur, de connaître précisément le nombre de cycles exécutées pour un programme à l'avance, même à partir du code assembleur.

Remarque 1

Pour plus de détails sur la fonction `sleep()`, vous pouvez vous référer à la documentation POSIX concernant l'en-tête `unistd.h`. Sur Linux, vous pouvez utiliser la commande `man 3 sleep` pour obtenir directement cette documentation.

Question 1.7

Cet algorithme fonctionne bien : on attend bien d'avoir une éprouvette libre avant d'y aller et d'y injecter une solution, puis on renvoie l'indice de l'éprouvette. De plus, on évite bien les problèmes liés aux threads concurrents : tous les problèmes évoqués dans les premières questions sont réglés ici (on reconnaît en partie la solution apportée à la **Question 1.3**).

Question 1.8

Bien que cette fonction est correcte comme expliqué dans la **Question 1.7**, cette approche a un impact négatif sur les autres threads car elle bloque tout autre appel à la fonction `déposer`, et même les autres processus car elle bloque toute autre utilisation du bras, car tout le corps de la boucle `while` est contenu dans la section critique d'un mutex. On a donc ici une mauvaise gestion de la concurrence.

Question 1.9

```c
int N = ...; // Nombre d'éprouvettes
pthread_mutex_t mutex_bras = PTHREAD_MUTEX_INITIALIZER;
sem_t sem_eprouvettes;

int déposer(struct formule *f, int *eprouvettes) {
  sem_wait(&sem_eprouvettes);
  pthread_mutex_lock(&mutex_bras);
  int i = trouver(eprouvettes, N);
  if (i == -1) {
    perror("État incohérent : une éprouvette devrait être libre");
    exit(1);
  }
  eprouvettes[i] = 1;
  aller(i);
  injecter(f);
  pthread_mutex_unlock(&mutex_bras);
  return i;
}

struct resultat récupérer(int pos, int *eprouvettes) {
  pthread_mutex_lock(&mutex_bras);
  aller(i);
  struct resultat res = analyser();
  eprouvettes[i] = 0;
  pthread_mutex_unlock(&mutex_bras);
  sem_post(&sem_eprouvettes);
  return res;
}

int main() {
  int *eprouvettes = calloc(sizeof(int), N);
  sem_init(&sem_eprouvettes, 0, N);
  // Appels parallèles à déposer et analyser
}
```

On a ajouté ici deux éléments : le tableau d'entiers `eprouvettes` et la sémaphore `sem_eprouvettes`. Le tableau `eprouvettes` désigne les éprouvettes, de telle sorte que `eprouvettes[i]` vaut `0` si l'éprouvette d'indice `i` est libre, et `1` si elle est occupée. On s'assure ensuite qu'un appel à la fonction `trouver` ne renverra jamais `-1` grâce à la sémaphore `sem_eprouvettes` qui compte le nombre d'éprouvettes libres à tout moment.

On suppose ici disposer d'une variable globale `N` indiquant le nombre total d'éprouvettes pour l'expérience.

Question 1.10

Pour assurer la synchronisation entre différents processus, il aurait fallu faire échanger l'état du tableau `eprouvettes` entre tous les processus, par exemple grâce à la fonction `pipe` et initialiser la sémaphore et le mutex pour permettre une utilisation inter-processus.

Question 1.11

Pour permettre l'utilisation de la fonction `déposer` par plusieurs threads d'un même programme en garantissant le bon fonctionnement du système, on doit utiliser au minimum un mutex, de la façon suivante :

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void déposer(struct formule *f) {
  int x, y;
  pthread_mutex_lock(&mutex);
  libreXY(&x, &y);
  allerX(x);
  allerY(y);
  injecter(f);
  pthread_mutex_unlock(&mutex);
}
```

Question 1.12

On peut utiliser des threads dans ce cas pour permettre l'utilisation des deux moteurs simultanément. On peut par exemple utiliser l'implémentation suivante :

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void allerXptr(int *x) {
  allerX(*x);
}

void déposer(struct formule *f) {
  int x, y;
  pthread_t thread;

  pthread_mutex_lock(&mutex);
  libreXY(&x, &y);

  pthread_create(&thread, NULL, &allerXptr, &x);
  allerY(y);
  pthread_join(thread, NULL);

  injecter(f);
  pthread_mutex_unlock(&mutex);
}
```

On utilise ici un thread pour lancer un appel à `allerX` de façon parallèle à l'appel à `allerY`. Un seul thread est alors nécessaire car, comme on peut le voir dans l'implémentation proposée, l'appel à `allerY` peut se faire directement.

Question 1.13

On suppose ici que le temps d'analyse d'un échantillon est constant, c'est-à-dire qu'il prend 5 UT.

- L'analyse de 5 commence au temps `T0` et est déjà en cours, on considère que le bras est commence donc en position 5. Elle terminera donc au temps `T0+6`.
- Le thread d'analyse de 3 commence au temps `T0+1`, le bras se déplace donc de 5 vers 3 en 20 UT après la fin de l'analyse de 6, soit en `T0+6`, et l'analyse de 3 débute au temps `T0+26`, et termine donc en `T0+31`.
- Le thread d'analyse de 7 commence au temps `T0+2`, le bras se déplace donc de 3 vers 7 en 4 UT dès que possible, c'est-à-dire à partir de `T0+31`. L'analyse de 7 débute donc au temps `T0+35`, et termine en `T0+40`.
- Le thread d'analyse de 9 commence au temps `T0+8`, le bras se déplace donc de 7 vers 9 en 2 UT dès que possible, c'est-à-dire à partir de `T0+40`. L'analyse de 9 débute donc au temps `T0+42`, et se termine en `T0+47`.

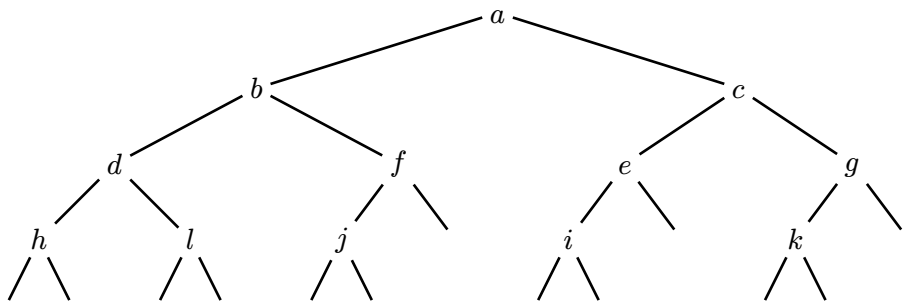
Question 1.14

> **Remarque 2**
>
> Je ne comprends comment l'énoncé présuppose que l'on peut savoir quelle analyse est en attente ou non.

Partie 2. Tableaux flexibles

Question 2.1

Voici l'arbre binaire correspondant à un tableau flexible de taille 12.



Question 2.2

```ocaml
let rec get n = function
  | E -> raise Not_found
  | N (_, x, _) when n = 0 -> x
  | N (l, _, r) -> get ((n - 1) / 2) (if n mod 2 = 0 then r else l)
```

Question 2.3

Montrons par induction que la hauteur d'un arbre a associé à tableau flexible de taille $n \in \mathbb{N}$, notée $h(a)$, est telle que : $h(a) = \lfloor \log_2(n+1) \rfloor$.

- Si l'arbre est égal à `E`, alors il possède 0 nœud et a une hauteur de 0, on a donc bien $h(\texttt{E}) = 0 = \lfloor \log_2(1) \rfloor$.
- On suppose que l'arbre a est de la forme $\texttt{N}(l, x, r)$. Comme les éléments de l sont les éléments du tableau d'indices de la forme $2i+1$, et ceux de r sont ceux d'indices $2i+2$. Donc, les arbres l et r ont au plus un de différence entre leurs nombres respectifs d'éléments.

  Supposons que l et r ont le même nombre d'éléments $\frac{n-1}{2}$: par hypothèses d'induction sur l et r, on a ainsi $h(l) = h(r) = \left\lfloor \log_2\left(\frac{n-1}{2} + 1\right) \right\rfloor = \left\lfloor \log_2\left(\frac{n-1+2}{2}\right) \right\rfloor = \lfloor \log_2(n+1) - 1 \rfloor = \lfloor \log_2(n+1) \rfloor - 1$ car $1 \in \mathbb{N}$. On a donc : $h(a) = \lfloor \log_2(n+1) \rfloor - 1 + 1 = \lfloor \log_2(n+1) \rfloor$. Le raisonnement est le même dans le cas où l et r ont exactement un élément d'écart.

Donc tout arbre associé à un tableau de taille $n \in \mathbb{N}$ a une hauteur égale à $\lfloor \log_2(n+1) \rfloor$, c'est-à-dire en $\mathcal{O}(\log(n))$.

Question 2.4

Pour un tableau de flexible de taille $n \in \mathbb{N}$, la fonction `liat` ne fera des appels récursifs que dans un seul sous-arbre à partir du nœud courant. Dans le pire cas, les appels récursifs parcourent donc exactement une fois la hauteur de l'arbre correspondant au tableau flexible. Or, d'après la **Question 2.3**, la hauteur d'un tableau flexible de taille n est $\mathcal{O}(\log(n))$, il y aura donc $\mathcal{O}(\log(n))$ appels récursifs à la fonction `liat`. De plus, hormis les appels récursifs, toutes les opérations de la fonction `liat` s'effectuent en temps constant. Donc, la complexité de la fonction `liat` est $\mathcal{O}(\log(n))$.

Question 2.5

Comme la complexité temporelle de `liat` est $\mathcal{O}(\log(n))$ d'après la question précédente, alors on a directement la terminaison de la fonction.

Il ne reste plus qu'à démontrer sa correction partielle, c'est-à-dire, en supposant que `liat` termine bien, que si t est un tableau flexible de taille $n \in \mathbb{N}^*$, alors `liat` renvoie bien un tableau flexible de taille $n-1$ contenant les $n-1$ premiers éléments de t.

On raisonne sur un appel à `liat t n` :

- si t est le tableau flexible vide, alors sa taille $n = 0$ on ne peut donc pas retirer l'un de ses éléments : ce cas est bien incohérent
- si t est un nœud dont les deux enfants sont des arbres vides, alors $n = 1$, et la fonction `liat` renvoie bien le seul tableau flexible de taille $n - 1 = 0$: l'arbre vide
- sinon, t un nœud dont au moins l'un des enfants est non vide, donc $n \geqslant 2$. Dans ce cas, si n est pair, alors le dernier élément est dans le sous-arbre gauche de t, car celui-ci est formé de tous les éléments d'indices impairs. Comme la structure de tableau flexible est définie récursivement, on peut poursuivre la recherche dans le sous-arbre gauche comprenant $\frac{n}{2}$ éléments, le sous-arbre droit reste alors identique. Par hypothèse de correction partielle sur l'appel récursif, on obtient alors bien le résultat demandé. On raisonne de la même façon dans le cas n impair.

Donc dans tous les cas, si t est un tableau flexible de taille $n \geqslant 1$, alors `liat t n` termine bien et renvoie un tableau flexible de taille $n-1$ et contenant les bons éléments.

Question 2.6

```ocaml
let rec snoc (n : int) (t : tree) (x : elt) : tree =
  match t with
  | E -> N (E, x, E)
  | N (l, y, r) ->
      if n mod 2 = 0 then N (l, y, snoc ((n / 2) - 1) r x)
      else N (snoc (n / 2) l x, y, r)
```

Question 2.7

```ocaml
let rec tail (t : tree) : tree =
  match t with
  | E | N (E, _, N (_, _, _)) -> assert false
  | N (E, _, E) -> E
  | N (N (ll, x, lr), _, r) -> N (r, x, tail (N (ll, x, lr)))
```

Dans cette fonction, pour les mêmes raisons qu'à la **Question 2.4**, on a bien une complexité temporelle de $\mathcal{O}(\log(n))$, dû au fait qu'il n'y a qu'un seul appel récursif par nœud et que la hauteur de l'arbre est logarithmique en sa taille.

Question 2.8

```ocaml
let rec cons (x : elt) (t : tree) : tree =
  match t with E -> N (E, x, E) | N (l, y, r) -> N (cons y r, x, l)
```

Pour les mêmes raisons que la question précédente, cette fonction a une complexité temporelle de $\mathcal{O}(\log(n))$.

Question 2.9

```ocaml
let rec of_sub_array (a : elt array) (m : int) (k : int) : tree =
  let n = Array.length a in
  if k >= n then E
  else
    N
      ( of_sub_array a (2 * m) (k + m),
        a.(k),
        of_sub_array a (2 * m) (k + (2 * m)) )

let of_array (a : elt array) : tree = of_sub_array a 1 0
```

Question 2.10

Intuitivement, on se dit qu'on peut construire un tableau flexible de taille n à partir de deux tableaux de taille $\lfloor n/2 \rfloor$, ce qui permettrait une complexité spatiale et temporelle en $\mathcal{O}(\log n)$. Il faut cependant ajuster les valeurs concrètes des tailles des sous-arbres gauche et droit, sachant que ceux-ci diffèrent en taille d'au plus 1. On définit donc une fonction auxiliaire qui pour n construit les tableaux flexibles de taille n et $n+1$.

```ocaml
let rec make_aux (n : int) (x : elt) : tree =
```

```
2     match n with
3     | 0 -> (E, N (E, x, E))
4     | n ->
5         let t, t' = make_aux x ((n - 1) / 2) in
6         if n mod 2 = 0 then (N (t', x, t), N (t', x, t'))
7         else (N (t, x, t), N (t', x, t))
8
9   let make (n : int) (x : elt) : tree = fst (make_aux x n)
10
11  let () =
12    assert (tree_eq (snoc 4 (liat 5 example_1) "e") example_1);
13    assert (tree_eq (liat 6 @@ snoc 5 example_1 "_") example_1);
14    assert (tree_eq (cons "a" @@ tail example_1) example_1);
15    assert (tree_eq (tail @@ cons "_" example_1) example_1);
16    assert (tree_eq example_1 @@ of_array [| "a"; "b"; "c"; "d"; "e" |]);
17    assert (tree_eq (make 75 "a") (of_array @@ Array.make 75 "a"))
```

On observe que $\lceil \log n \rceil$ est un variant de l'algorithme, ce qui garantit la terminaison de l'algorithme et fournit la complexité attendue, en remarquant que les opérations au cours d'un appel se font en temps constant. Cela fournit également la complexité en espace en $\mathcal{O}(\log n)$, puisque chaque appel réalise un nombre constant d'allocations (en fait, trois : celle de la paire d'arbre, et les deux nœuds … mais c'est du détail).

Concernant la correction, il faut distinguer selon la parité de n:

- si $n = 2i$, l'appel récursif se fait sur $i - 1$. Dans ce cas, l'arbre de taille n se construit en utilisant les arbres de taille i (à gauche) et $i - 1$ (à droite), et l'arbre de taille $n + 1$ avec deux fois l'arbre de taille i;
- sinon $n = 2i + 1$, l'appel récursif se fait sur i. L'arbre de taille n se fait avec deux fois l'arbre de taille i, et l'arbre de taille $n + 1$ se fait avec les arbres de taille $i + 1$ et i.

Partie 3. Gestion d'un concours de recrutement

Question 3.1

L'intérêt d'avoir `cid` comme clef primaire de `AgrégExtInfo` est de s'assurer qu'un candidat ne peut pas s'inscrire plusieurs fois, que ce soit dans une ou plusieurs académies différentes.

Question 3.2

```sql
SELECT aid, COUNT(*)
FROM AgrégExtInfo
GROUP BY aid;
```

Question 3.3

Voici la requête en langage SQL :

```sql
SELECT c.nom, aei.aid, cnsi.aid
FROM Candidat c
```

```sql
JOIN AgrégExtInfo aei
JOIN CapesNSI cnsi
ON c.cid = aei.cid
  AND c.cid = cnsi.cid
WHERE aei.aid <> cnsi.aid;
```

et la même requête en algèbre relationnelle :

$$\pi_{\text{cid, aid1, aid2}}\Big(\sigma_{\text{aid1} \neq \text{aid2}}(\text{Candidat} \bowtie \rho_{\text{aid} \to \text{aid1}}(\text{AgrégExtInfo}) \bowtie \rho_{\text{aid} \to \text{aid2}}(\text{CapesNSI}))\Big)$$

Question 3.4

```sql
SELECT c.profession, COUNT(*)
FROM Candidat c
JOIN AgrégExtInfo aei
ON c.cid = aei.cid
WHERE c.cid NOT IN (
  SELECT cid
  FROM CapesNSI
)
GROUP BY c.profession;
```

Question 3.5

```sql
SELECT COUNT(*) / (
  SELECT COUNT(*)
  FROM Candidat c
  JOIN AgrégExtInfo aei
  ON c.cid = aei.cid
  WHERE c.ep = "Étude de cas informatique"
    AND c.genre = "F"
)
FROM Candidat c
JOIN AgrégExtInfo aei
ON c.cid = aei.cid
WHERE c.ep = "Étude de cas informatique";
```

Question 3.6

Voici la requête en langage SQL :

```sql
SELECT c1.nom
FROM Candidat c1
JOIN ListeAmén la1
JOIN Aménagement a1
ON c1.cid = la1.cid
  AND la1.amid = a1.amid
WHERE a1.amén = "Salle isolée"
```

```
    AND "Tiers-temps" NOT IN (
      SELECT a2.amén
      FROM Candidat c2
      JOIN ListeAmén la2
      JOIN Aménagement a2
      ON c2.cid = la2.cid
        AND la2.amid = a2.amid
      WHERE c1.cid = c2.cid
    );
```

et la même requête en algèbre relationnelle :

$$\pi_{\text{nom}}(\sigma_{\text{amén} = \text{"Salle isolée"}}(\text{Candidat} \bowtie \text{ListeAmén} \bowtie \text{Aménagement})) \setminus \pi_{\text{nom}}(\sigma_{\text{amén} = \text{"Tiers-temps"}}(\text{Candidat} \bowtie \text{ListeAmén} \bowtie \text{Aménagement}))$$

Question 3.7

Voici la requête SQL :

```sql
SELECT ac.nom
FROM Académie ac
WHERE ac.nom NOT IN (
  SELECT ac.nom
  FROM Académie ac
  JOIN AgrégExtInfo aei
  JOIN ListeAmén la
  JOIN Aménagement a
  ON ac.aid = aei.aid
    AND aei.cid = la.cid
    AND la.amid = a.amid
  WHERE amén = "Tiers-temps"
);
```

et la même requête en algèbre relationnelle :

$$\pi_{\text{nom}}(\text{Académie}) \setminus \pi_{\text{nom}}(\sigma_{\text{amén} = \text{"Tiers-temps"}}(\text{Académie} \bowtie \text{AgrégExtInfo} \bowtie \text{ListAmén} \bowtie \text{Aménagement}))$$

Question 3.8

```sql
WITH salles_isolees_academie AS (
  SELECT ac.nom AS ac_nom, COUNT(*) AS nb_salles_isolees
  FROM Académie ac
  JOIN AgrégExtInfo aei
  JOIN ListeAmén la
  JOIN Aménagement a
  ON ac.aid = aei.aid
    AND aei.cid = la.cid
    AND la.ami = a.amid
  WHERE a.amén = "Salle isolée"
```

```
11    GROUP BY ac.amid
12  ) SELECT ac_nom
13  FROM salles_isolees_academies
14  WHERE nb_salles_isolees = (
15    SELECT MAX(nb_salles_isolees)
16    FROM salles_isolees_academie
17  );
```

Question 3.9

```sql
SELECT c.nom, note-ep1 + note-ep2 + note-ep3 AS note
FROM Candidats c
JOIN AgrégExtInfo aei
ON c.cid = aei.cid
ORDER BY note DESC
LIMIT 42;
```

Question 3.10

```sql
SELECT AVG(note-ep1 + note-ep2 + note-ep3) AS moyenne
FROM Candidats c
JOIN AgrégExtInfo aei
JOIN ListAmén la1
JOIN ListAmén la2
JOIN Aménagement a1
JOIN Aménagement a2
ON c.cid = aei.cid
  AND c.cid = la1.cid
  AND c.cid = la2.cid
  AND la1.amid = a1.amid
  AND la2.amid = a2.amid
WHERE la1.amid <> la2.amid;
```

Question 3.11

On peut modéliser une telle situation de la manière suivante : on crée une table « Concours » contenant deux champs, une clef primaire entière `coid` contenant un identifiant unique à chaque concours, et une chaîne de caractères `nom` contenant le nom du concours. On ajoute alors une seconde table « Inscription » contenant deux champs : une clef étrangère `cid` vers la table « Candidats », et une clef étrangère `coid` vers la table « Concours ». Ainsi, ajouter un concours ne demande plus de créer une nouvelle table mais une nouvelle ligne dans la table « Concours », et on peut y inscrire les candidates en passant par la table « Inscription ».

On peut alors étendre cette modélisation pour ajouter des informations supplémentaires, par exemple un champs `option` pour les concours contenant un choix entre plusieurs options, voire une table supplémentaire « Notes » pour stocker les résultats des candidates aux différentes épreuves d'un concours.

Question 3.12

Avec la modélisation proposée à la question précédente, on peut alors écrire :

```sql
SELECT c.nom, COUNT(*) AS nombre_concours
FROM Inscriptions i
GROUP BY c.cid
HAVING nombre_concours >= ALL(
  SELECT COUNT(*)
  FROM Inscriptions i
  GROUP BY c.cid
);
```

Question 3.13

Remarque 3

Pour cette question et les suivantes, il n'est pas précisé ici de quelle manière vérifier les données. Le choix ici a été fait de renvoyer un booléen indiquant si les données sont conformes, mais on aurait pu imaginer mettre des `assert` à la place des `return False` et alors ne rien renvoyer.

```python
def verif_nblc(N, t_oral):
    if len(t_oral) != 3 * N:
        return False
    for ligne in t_oral:
       if len(ligne) != 10:
          return False
    return True
```

Question 3.14

```python
def verif_temps(oral):
    for ligne in oral:
        epreuve = ligne[1]
        convoc = ligne[2])
        prep = ligne[3]
        passage = ligne[4]
        fin = ligne[5]
        if not (prep - convoc == 15 and fin - passage == 60):
            return False
        if epreuve == "TP" and passage - prep != 300:
            return False
        if epreuve in ["Leçon", "Modélisation"] and passage - prep != 240:
            return False
    return True
```

Question 3.15

```python
def verif_nbj(oral, nb_jury):
    # Dictionnaire où les clefs sont (Jour, Hpass)
    # et les valeurs le nombre de candidats pour cet horaire
    nb_candidats = {}
    for ligne in oral:
        jour = ligne[0]
        passage = ligne[4]
        clef = (jour, passage)
        if not clef in nb_candidats:
            nb_candidats[clef] = 1
        else:
            nb_candidats[clef] += 1
    for nb in nb_candidats.values():
        if nb > nb_jury:
            return False
    return True
```

Question 3.16

```python
def verif_ep(oral):
    # Dictionnaire où les clefs sont cid
    # et les valeurs un dictionnaire des épreuves
    # à leurs dates
    candidats = {}

    # On remplit tout d'abord la variable candidats
    for ligne in oral:
        cid = ligne[7]
        epreuve = ligne[1]
        jour = ligne[0]
        if cid in candidats:
            # On s'assure qu'aucun candidat ne passe
            # deux fois la même épreuve
            if epreuve in candidats[cid]:
                return False
            if jour in candidats[cid].values():
                return False
        else:
            candidats[cid] = {}
            candidats[cid][epreuve] = jour

    # On vérifie maintenant que chaque candidat passe
    # bien trois épreuves. On sait alors qu'elles sont
    # différentes car on a déjà repéré les doublons.
```

```
26      for epreuves in candidats.values():
27          if len(epreuves) != 3:
28              return False
29
30      return True
```

Question 3.17

```python
def dict_verif_nomno(dict_oral):
    # Dictionnaire où les clefs sont cid
    # et les valeurs les couples (Nom, Prénom)
    noms = {}

    for oral in dict_oral:
      cid = oral["cid"]
      nom = oral["Nom"]
      prenom = oral["Prénom"]
      if cid in noms:
        (nom2, prenom2) = noms[cid]
        if nom != nom2 or prenom != prenom2:
          return False
      else:
        noms[cid] = (nom, prenom)

    return True
```

Question 3.18

Les questions étant très similaires, on s'inspire de la **Question 3.16**.

```python
def dict_verif_ep(dict_oral):
    # Dictionnaire où les clefs sont cid
    # et les valeurs un dictionnaire des épreuves
    # à leurs dates
    candidats = {}

    # On remplit tout d'abord la variable candidats
    for oral in dict_oral:
        cid = oral["cid"]
        epreuve = oral["Ép"]
        jour = oral["Jour"]
        if cid in candidats:
            # On s'assure qu'aucun candidat ne passe
            # deux fois la même épreuve
            if epreuve in candidats[cid]:
                return False
```

```
17
18              # On s'assure qu'aucun candidat ne passe
19              # deux fois le même jour
20              if jour in candidats[cid].values():
21                  return False
22          else:
23              candidats[cid] = {}
24              candidats[cid][epreuve] = jour
25
26      # On vérifie maintenant que chaque candidat passe
27      # bien trois épreuves. On sait alors qu'elles sont
28      # différentes et qu'elles ont lieu sur des jours
29      # différents car on a déjà repéré les doublons.
30      for epreuves in candidats.values():
31          if len(epreuves) != 3:
32              return False
33
34      return True
```

Question 3.19

L'utilisation des dictionnaires ici est avantageuse par rapport à l'approche utilisée précédemment car :

- il est plus simple pour un humain de faire référence à un champ précis par son nom que par son indice ;
- les dictionnaires s'interfacent plus facilement avec d'autres librairies, comme celles manipulant des fichiers CSV ou des bases de données, car les noms des colonnes sont déjà indiquées.

Cependant, celle-ci est désavantageuse sur certains autres points :

- les dictionnaires sont des structures plus complexes que les tableaux, que ce soit en mémoire ou en temps : il y aura un léger surcoût en pratique même si asymptotiquement les tableaux et les dictionnaires peuvent être équivalents en temps et en mémoire.

Question 3.20

L'utilisation d'un fichier CSV est avantageuse par rapport à une base de données sur certains points :

- un fichier CSV est bien plus simple à mettre en place qu'une base de données avec un SGBD ;
- la lecture et la modification des données à la main est réalisable directement depuis n'importe quel logiciel de traitement de texte.

Cependant, l'utilisation d'une base de données présente aussi des avantages :

- dans le cas d'un très grand nombre de données, le SGBD traitera les requêtes bien plus rapidement qu'un fichier, celui-ci étant optimisé pour les données structurées contrairement au systèmes de fichiers.

Question 3.21

```python
def heure_to_int(horaire: str) -> int:
    heures, minutes = horaire.split("h")
    heures = int(heures)
    if minutes == "":
```

```
5           minutes = 0
6       else:
7           minutes = int(minutes)
8       if heures < 0 or heures > 23:
9           raise ValueError("L'heure donnée n'est pas comprise entre 0h et 24h")
10      if minutes < 0 or minutes > 59:
11          raise ValueError("Les minutes données ne sont pas cohérentes")
12      return heures * 60 + minutes
13
14  def date_to_jour(date: str) -> int:
15      jour, _ = date.split("/")
16      jour = int(jour)
17      if jour < 1 or jour > 31:
18        raise ValueError("Le jour donné n'est pas cohérent")
19      return jour
20
21  def clean_dict(t_dict_oral):
22      dict_oral = []
23      for oral in t_dict_oral:
24          clean_oral = {}
25          clean_oral["Jour"] = date_to_jour(oral["Jour"])
26          for clef in ["Hconov", "Hdéb", "Hpass", "Hfin"]:
27              clean_oral[clef] = heures_to_int(oral[clef])
28          for clef in ["Ép", "Jury", "Nom", "Prénom"]
29              clean_oral[clef] = oral[clef]
30          clean_oral["cid"] = int(oral["cid"])
31          dict_oral.append(clean_oral)
32      return dict_oral
```