Correction du DS 1

Julien Reichert

Partie 1

Exercice 1.1

```
def min(l):
    assert l != []
    rep = l[0]
    for i in range(1, len(l)):
        if l[i] < rep:
            rep = l[i]
    return rep
```

Exercice 1.2

On encadre la phase décroissante et la phase croissante par les variables `deb` et `fin`, sachant qu'on identifie la phase en comparant une valeur à une valeur voisine. La limite entre les phases donne le minimum.

Pour simplifier, on commence par vérifier que la liste n'est pas de taille 1 ou strictement décroissante ou strictement croissante par deux comparaisons de plus (une liste vide déclenchera une erreur, ce qui est normal).

```
def min_dicho(l):
    if len(l) == 1 or l[0] < l[1]:
        return l[0]
    if l[-1] < l[-2]:
        return l[-1]
    deb = 0
    fin = len(l)-1
    while deb < fin:
        mil = (deb + fin)//2
        if l[mil] < l[mil-1]:
            if l[mil] < l[mil+1]:
                return l[mil]
            else:
                deb = mil + 1
        else:
            fin = mil - 1
    return l[deb]
```

L'intervalle de recherche perdant plus de la moitié de sa taille à chaque tour de boucle, la complexité est bien logarithmique.

Pour se convaincre que la fonction est correcte, on peut confirmer qu'à chaque tour de boucle, le minimum est entre l'indice `deb` et l'indice `fin` car on bascule au-delà de `mil` (strictement) en restant du bon côté.

Exercice 1.3

La croissance des listes permet d'utiliser un algorithme efficace inspiré de la fonction de fusion du tri du même nom.

```
def un_commun(l, ll):
    n, nn = len(l), len(ll)
    i, ii = 0, 0
    while i < n and ii < nn:
        if l[i] == ll[ii]:
            return True
        elif l[i] < ll[ii]:
            i += 1
        else:
            ii += 1
    return False
```

On commence par prouver la terminaison : `nn + n - (ii + i)` est un variant qui diminue d'un à chaque tour de boucle. Ceci permet de déduire que le nombre de tours de boucle est majoré par la somme des tailles des listes, et que la complexité est linéaire en cette somme puisque le corps de boucle est en $\mathcal{O}(1)$.

Exercice 1.4

Le plus grand produit est le maximum entre le produit des deux plus petits nombres et celui des deux plus grands nombres. Il ne peut être strictement négatif que s'il n'y a que deux éléments dans la liste, tous non nuls et de signe opposé. On cherche donc ces quatre (ou moins si la liste est trop courte) valeurs, le tout en un seul passage.

Pour information, il existe une optimisation (qui reste en temps linéaire mais avec un meilleur facteur constant) de la recherche des deux plus grands éléments, cette optimisation n'est pas traitée ici.

```
def plus_grand_produit(l):
    assert len(l) >= 2, "Liste trop courte"
    max1 = l[l[0] < l[1]] # Oh la belle astuce pour éviter le copier-coller !
    max2 = l[l[0] > l[1]]
    min1 = l[l[0] > l[1]]
    min2 = l[l[0] < l[1]]
    for i in range(2, len(l)):
        if l[i] > max1:
            max2 = max1
            max1 = l[i]
        elif l[i] > max2:
            max2 = l[i]
        if l[i] < min1: # pas elif
            min2 = min1
            min1 = l[i]
        elif l[1] < min2:
            min2 = l[i]
    return max(max1 * max2, min1 * min2)
```

Exercice 1.5

Bien entendu, il faut commencer par récupérer une version triée de l'entrée. Une astuce possible revient à prendre un tri non en place (tri fusion par exemple) et à remplacer l'opérateur de comparaison utilisé par une fonction `compare`, qui peut alors être mise en argument de la fonction de tri, avec pour valeur par défaut le test qu'on aurait employé.

Ensuite, vu qu'on a payé au moins du $n \log(n)$, on peut faire un parcours supplémentaire de la version triée et voir si on a à chaque fois des égalités.

```
def merge(l1, l2, comparaison):
    n1, n2, i1, i2 = len(l1), len(l2), 0, 0
    l = [None] * (n1+n2)
    while i1 < n1 and i2 < n2:
        if comparaison(l1[i1], l2[i2]):
            l[i1+i2] = l1[i1]
            i1 += 1
        else:
            l[i1+i2] = l2[i2]
            i2 += 1
    for i in range(i1, n1): # vide si i1 = n1, termine l
            l[i+i2] = l1[i]
    for i in range(i2, n2): # vide si i2 = n2, termine l
            l[i1+i] = l2[i]
    return l

def merge_sort(l, comparaison=lambda x, y : x < y):
    n = len(l)
    if n <= 1:
        return l
    l1 = merge_sort(l[:n//2], comparaison)
    l2 = merge_sort(l[n//2:], comparaison)
    return merge(l1, l2, comparaison)

def comp_class(a, b):
    return a[1] > b[1] # merge_sort trie dans l'ordre croissant !

def classement(liste):
    ordre = merge_sort(liste, comp_class)
    place = 1
    valeur_precedente = None
    for i in range(len(ordre)):
        (nom, valeur) = ordre[i]
        if valeur_precedente != valeur:
            place = i + 1 # On réinitialise la place qui ne peut être que l'indice actuel plus un
            valeur_precedente = valeur
        print(place, nom)
```

Partie 2

Exercice 2.1

Clés (primaires) : `id` pour `ETUDIANTS`, `id` pour `EXAMENS` et `etudiant, examen` pour `NOTES`. Clés étrangères : `etudiant` dans `NOTES` vers `id` dans `ETUDIANTS` et `examen` dans `NOTES` vers `id` dans `EXAMENS`.

Exercice 2.2

```
SELECT date FROM EXAMENS
```

Exercice 2.3

```
SELECT DISTINCT matiere FROM EXAMENS JOIN NOTES ON id=examen WHERE note=20
```

Exercice 2.4

```
SELECT DISTINCT matiere FROM EXAMENS
JOIN NOTES ON EXAMENS.id = examen
JOIN ETUDIANTS ON etudiant = ETUDIANTS.id
WHERE classe='PCSI1'
```

Exercice 2.5

```
SELECT COUNT(DISTINCT prenom) FROM ETUDIANTS
GROUP BY classe
ORDER BY COUNT(DISTINCT prenom) DESC
LIMIT 1
```

Exercice 2.6

```
SELECT prenom FROM ETUDIANTS
GROUP BY prenom
ORDER BY COUNT(*) DESC
LIMIT 1
```

Version qui tient compte des égalités :

```
SELECT prenom FROM ETUDIANTS
GROUP BY prenom
HAVING COUNT(*) =
(
  SELECT COUNT(*) FROM ETUDIANTS
  GROUP BY prenom
  ORDER BY COUNT(*) DESC
  LIMIT 1
)
```

Exercice 2.7

```
SELECT classe FROM ETUDIANTS
WHERE prenom = 'Théo'
GROUP BY classe
ORDER BY COUNT(*) DESC
LIMIT 1
```

Il est plus simple d'utiliser `WHERE`, mais dans ce qui suit une idée alternative raccourcira l'écriture...

Version qui tient compte des égalités :

```
SELECT classe FROM ETUDIANTS
GROUP BY classe
HAVING SUM(prenom = 'Théo') =
(
  SELECT SUM(prenom = 'Théo') FROM ETUDIANTS
  GROUP BY classe
  ORDER BY SUM(prenom = 'Théo') DESC
  LIMIT 1
)
```

Exercice 2.8

```
SELECT prenom FROM ETUDIANTS JOIN NOTES ON id = etudiant
WHERE examen = 1
GROUP BY prenom
ORDER BY AVG(note) DESC LIMIT 1
```

Version qui tient compte des égalités :

```
SELECT prenom FROM ETUDIANTS JOIN NOTES ON id = etudiant
WHERE examen = 1
GROUP BY prenom
HAVING AVG(note) =
(
  SELECT AVG(note) FROM ETUDIANTS JOIN NOTES ON id = etudiant
  WHERE examen = 1
  GROUP BY prenom
  ORDER BY AVG(note) DESC LIMIT 1
)
```

Exercice 2.9

```
SELECT note FROM NOTES JOIN EXAMENS ON examen = id
WHERE matiere = "Mathématiques" AND date = "2023-09-09"
ORDER BY note LIMIT 5
```

Exercice 2.10

```
SELECT note, COUNT(*) AS effectif FROM NOTES
GROUP BY note
ORDER BY note
```

Exercice 2.11

```
def requete_2_2():
    return [enreg[1] for enreg in EXAMENS]
```

Exercice 2.12

```
def requete_2_5():
    dico = dict()
    for enreg in ETUDIANTS:
        classe = enreg[3]
        prenom = enreg[2]
        if classe not in dico:
            dico[classe] = dict()
        if prenom not in dico[classe]:
            dico[classe][prenom] = 1
    taillemax = 0
    for classe in dico:
        if len(dico[classe]) > taillemax:
            taillemax = len(dico[classe])
    return taillemax
```

Exercice 2.13

Pour faire bonne mesure, on renverra ici la liste des classes maximisant le nombre de Théo, en tenant compte des égalités.

```
def requete_2_7():
    dico = dict()
    for enreg in ETUDIANTS:
        classe = enreg[3]
        prenom = enreg[2]
        if classe not in dico:
            dico[classe] = 0 # potentiellement aucun Théo tout court
        if prenom == "Théo":
            dico[classe] += 1
    theomax = 0
    classesmax = [] # si pas de Théo, toutes les classes existantes seront dans la liste
    for classe in dico:
        if dico[classe] > theomax:
            classesmax = [classe]
            theomax = dico[classe]
        elif dico[classe] == theomax:
            classesmax.append(classe)
    return classesmax
```

Exercice 2.14

On ne fait pas de restriction sur les notes (mais avec la restriction, le dictionnaire peut être remplacé par une liste, et on pourrait tolérer que les notes n'ayant jamais été attribuées soient incorporées à la liste et associées à la valeur 0).

```
def requete_2_10()
    dico = dict()
    for note in NOTES:
        if note not in dico:
            dico[note] = 1
        else:
            dico[note] += 1
    rep = []
    for note in sorted(dico.keys()):
        rep.append((note, dico[note]))
    return rep
```

Partie 3

Exercice 3.1

```
SELECT MAX(points) FROM QUESTION WHERE id_ds = 1
```

Exercice 3.2

```
SELECT COUNT(*) FROM NOTE WHERE id_etudiant = 1 AND pourcentage = 1.0
```

Exercice 3.3

Questions traitées :

```
SELECT COUNT(DISTINCT code_question) FROM NOTE WHERE id_ds = 1
```

Questions non traitées, deux options :

```
SELECT COUNT(*) FROM QUESTION
WHERE id_ds = 1
AND code_question NOT IN
(
  SELECT code_question FROM NOTE
  WHERE id_ds = 1
)
```

Ci-avant : sous-requête avec `IN` et test d'appartenance ; ci-après : sous-requêtes et différence de cardinaux.

```
SELECT
(SELECT COUNT(*) FROM QUESTION WHERE id_ds = 1)
-
(SELECT COUNT(DISTINCT code_question) FROM NOTE WHERE id_ds = 1)
```

Exercice 3.4

```
SELECT MAX(points*pourcentage)
FROM NOTE
JOIN QUESTION ON NOTE.id_ds = QUESTION.id_ds AND NOTE.code_question = QUESTION.code_question
```

Exercice 3.5

```
SELECT SUM(points*pourcentage) FROM NOTE
JOIN QUESTION ON NOTE.id_ds = QUESTION.id_ds AND NOTE.code_question = QUESTION.code_question
WHERE id_etudiant = 1
AND id_ds = 1
```

Exercice 3.6

```
SELECT nom_partie FROM NOTE
JOIN QUESTION ON NOTE.id_ds = QUESTION.id_ds AND NOTE.code_question = QUESTION.code_question
WHERE id_etudiant = 1
AND id_ds = 1
GROUP BY nom_partie
ORDER BY SUM(points*pourcentage) DESC
LIMIT 1
```

Version tenant compte des égalités :

```
SELECT nom_partie FROM NOTE
JOIN QUESTION ON NOTE.id_ds = QUESTION.id_ds AND NOTE.code_question = QUESTION.code_question
WHERE id_etudiant = 1
AND id_ds = 1
GROUP BY nom_partie
HAVING SUM(points*pourcentage) =
(
  SELECT SUM(points*pourcentage) FROM NOTE
  JOIN QUESTION ON NOTE.id_ds = QUESTION.id_ds AND NOTE.code_question = QUESTION.code_question
  WHERE id_etudiant = 1
  AND id_ds = 1
  GROUP BY nom_partie
  ORDER BY SUM(points*pourcentage) DESC
  LIMIT 1
)
```

Exercice 3.7

```
SELECT prenom, nom, SUM(points*pourcentage) AS note FROM NOTE
JOIN QUESTION ON NOTE.id_ds = QUESTION.id_ds AND NOTE.code_question = QUESTION.code_question
JOIN ETUD ON NOTE.id_etudiant = ETUD.id_etudiant
WHERE id_ds = 1
GROUP BY id_etudiant
ORDER BY note DESC
```

Exercice 3.8

C'est exactement la même requête mais le calcul renommé en l'attribut `note` se termine en divisant par `SUM(points)`.