Correction du DS 2

Julien Reichert

Partie 1

Exercice 1.1

Pour la table `INSCRIPTION`, on ne souhaite pas avoir à un même tournoi deux personnes ayant le même numéro (peu pratique vu qu'on a créé l'attribut en question exprès pour la concision lors de jointures avec d'autres tables), le même nom (sinon le classement est ambigu) et le même identifiant d'utilisateur (vu qu'il est de toute façon personnel).

Pour la table `SERIE`, on a naturellement la clé `(id, serie, numero)` car on n'entre qu'un résultat par joueur à chaque série, mais aussi la clé `(id, serie, numtable, place)` car un seul joueur peut s'asseoir à une même place à une série donnée.

Exercice 1.2

```
SELECT COUNT(*)
FROM INSCRIPTION AS I
JOIN TOURNOI AS T ON I.id = T.id
WHERE titre = "Tournoi de Nancy"
```

Exercice 1.3

```
SELECT SUM(resultat)
FROM INSCRIPTION AS I
JOIN SERIE AS S ON I.id = S.id AND I.numero = S.numero
WHERE Nom = "Julien Reichert" AND I.id=5
```

Exercice 1.4

```
SELECT MAX(resultat)
FROM TOURNOI AS T
JOIN SERIE AS S ON T.id = S.id
WHERE club=2
```

Exercice 1.5

```
SELECT SUM(resultat) AS Total
FROM SERIE
WHERE id=2
GROUP BY numero
ORDER BY Total DESC LIMIT 1
```

Exercice 1.6

Une version plus rigoureuse utilisant `HAVING` est possible pour afficher les égalités, mais plus longue. L'exercice 1.8 se charge de la gestion des égalités, de toute manière.

```
SELECT utilisateur
FROM INSCRIPTION AS I
JOIN SERIE AS S ON I.id = S.id AND I.numero = S.numero
GROUP BY utilisateur
ORDER BY SUM(resultat) DESC LIMIT 1
```

Exercice 1.7

```
SELECT nom
FROM INSCRIPTION AS I
JOIN SERIE AS S ON I.id = S.id AND I.numero = S.numero
WHERE I.Id=t AND (parti = 0 OR parti >= n)
GROUP BY I.numero, nom
ORDER BY SUM(resultat) DESC, SUM(gagnes) DESC, SUM(perdus)
```

Exercice 1.8

```
SELECT nom, COUNT(*) AS participations
FROM INSCRIPTION
GROUP BY nom
HAVING COUNT(*) =
(
  SELECT COUNT(*) FROM INSCRIPTION GROUP BY nom ORDER BY COUNT(*) DESC LIMIT 1
)
```

Partie 2

Exercice 2.1

On a successivement les termes $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

Exercice 2.2

Le temps de vol en altitude est 6 et l'altitude maximale est 88.

Si u_0 est pair, le temps de vol est nécessairement 1 car $u_1 < u_0$, étant sa moitié.

Une fois que $u_n = 1$, la suite boucle sur les trois valeurs $1, 4, 2$ dans cet ordre.

Exercice 2.3

```
def terme_suivant(un):
    if un % 2 == 0:
        return un // 2
    else:
        return 3 * un + 1
```

Exercice 2.4

```
def temps_vol_altitude(N):
    un = N
    indice = 0
    while un >= N:
        un = terme_suivant(un)
        indice += 1
    return indice
```

Exercice 2.5

Version avec un dictionnaire :

```
def tous_temps_vol(N):
    dico = {1 : 0}
    def aux(un):
        if un not in dico:
            dico[un] = 1 + aux(terme_suivant(un))
        return dico[un]
    reponse = [None, 0]
    for i in range(2, N+1):
        reponse.append(aux(i))
    return reponse
```

Version plus compliquée avec une liste :

```
def tous_temps_vol2(N):
    reponse = [None for i in range(N+1)]
    reponse[1] = 0
    def aux(un):
        if un > N:
            return 1 + aux(terme_suivant(un))
        if reponse[un] is None:
            reponse[un] = 1 + aux(terme_suivant(un))
        return reponse[un]
    for i in range(2, N+1):
        reponse[i] = aux(i)
    return reponse
```

Exercice 2.6

Seulement la version avec un dictionnaire, l'autre étant analogue :

```
def toutes_altitudes_maximales(N):
    dico = { 1 : 1 }
    def aux(un):
        if un not in dico:
            dico[un] = max(un, aux(terme_suivant(un))
        return dico[un]
    reponse = [None, 1]
    for i in range(2, N+1):
        reponse.append(aux(i))
    return reponse
```

Partie 3

Exercice 3.1

```
def present(s, m):
    for i in range(len(s)): # Pas de risque de débordement avec une tranche !
        if s[i:i+len(m)] == m:
            return True
    return False
```

Exercice 3.2

Dans le pire des cas, on extrait toutes les tranches possibles et les comparaisons échouent. La complexité est linéaire en le produit des tailles des chaînes au vu du nombre de tours de boucle et de la complexité linéaire en la taille de `m` du corps des boucles.

Exercice 3.3

```
def present2(s, m):
    n = len(s)
    k = len(m)
    for i in range(n-k+1): # Accélération supplémentaire : on ne teste pas s'il n'y a plus la place.
        j = 0
        while j < k and s[i+j] == m[j]:
            j += 1
        if j == k:
            return True
    return False
```

Exercice 3.4

Dans le meilleur des cas, la première correspondance est la bonne et la complexité est de l'ordre de la taille de `m`.

Une différence avec la fonction précédente : dans le meilleur des cas parmi ceux où la réponse est `False`, le premier caractère de `m` est absent de `s` et chaque boucle s'arrête au premier tour, ce qui donne une complexité de l'ordre de la taille de `s` pour un gain considérable.

Exercice 3.5

Tour de boucle où `i` vaut `0` (début sur un `'C'`) :
- j = 0 : `s[0+0] == m[0] == 'C'`, on continue ;
- j = 1 : `s[0+1] == m[1] == 'H'`, on continue ;
- j = 2 : `s[0+2] == m[2] == 'E'`, on continue ;
- j = 3 : `s[0+3] != m[3]`, on sort.

`j` vaut `3` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `1` (début sur un `'H'`) :
- j = 0 : `s[1+0] != m[0]`, on sort.

`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `2` (début sur un `'E'`) :
- j = 0 : `s[2+0] != m[0]`, on sort.

`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `3` : (début sur un `'R'`) :
— `j` = 0 : `s[3+0] != m[0]`, on sort.
`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `4` (début sur un `'C'`) :
— `j` = 0 : `s[4+0] == m[0] == 'C'`, on continue ;
— `j` = 1 : `s[4+1] == m[1] == 'H'`, on continue ;
— `j` = 2 : `s[4+2] == m[2] == 'E'`, on continue ;
— `j` = 3 : `s[4+3] != m[3]`, on sort.
`j` vaut `3` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `5` (début sur un `'H'`) :
— `j` = 0 : `s[5+0] != m[0]`, on sort.
`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `6` (début sur un `'E'`) :
— `j` = 0 : `s[6+0] != m[0]`, on sort.
`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `7` : (début sur un `'R'`) :
— `j` = 0 : `s[7+0] != m[0]`, on sort.
`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `8` : (début sur un `' '`) :
— `j` = 0 : `s[8+0] != m[0]`, on sort.
`j` vaut `0` et pas `4`, on ne renvoie pas `True` ici.

Tour de boucle où `i` vaut `9` (début sur un `'C'`) :
— `j` = 0 : `s[4+0] == m[0] == 'C'`, on continue ;
— `j` = 1 : `s[4+1] == m[1] == 'H'`, on continue ;
— `j` = 2 : `s[4+2] == m[2] == 'E'`, on continue ;
— `j` = 3 : `s[4+3] == m[3] == 'Z'`, on continue ;
— `j` = 4 : on sort.
`j` vaut `4`, on renvoie `True`.

Exercice 3.6

```
def precalcule(m):
    k = len(m)
    dicos = [dict() for i in range(k)]
    dicos[0][m[0]] = 0
    for i in range(k-1):
        for cle in dicos[i]:
            dicos[i+1][cle] = dicos[i][cle] + 1
        dicos[i+1][m[i+1]] = 0
    return dicos
```

Exercice 3.7

```
def boyer_moore(s, m):
    dicos = precalcule(m)
    n = len(s)
    k = len(m)
    i = 0
```

```
    while i < n - k + 1:
        j = k - 1
        while j > -1 and s[i+j] == m[j]:
            j -= 1
        if j == -1:
            return True
        elif s[i+j] in dicos[j]:
            i += dicos[j][s[i+j]]
        else:
            i += j+1
    return False
```

Exercice 3.8

```
def hache_suivant(s, i, h, k, c):
    return (h * 97 - ord(s[i]) * c + ord(s[i+k])) % 997
```

Exercice 3.9

```
def hache_et_97k(s): # En pratique c'est cette fonction qu'on utilisera pour gagner un peu de temps !
    rep = 0
    c = 1
    for i in range(len(s)-1, -1, -1):
        rep += ord(s[i]) * c
        c *= 97
    return rep % 997, c

def hache(s): # Mais c'est cette fonction que l'énoncé demandait d'écrire !
    return hache_et_97k(s)[0]
```

Exercice 3.10

```
def rabin_karp(s, m):
    n = len(s)
    k = len(m)
    hm, c = hache_et_97k(m)
    hs = hache(s[:k])
    for i in range(n-k+1):
        if hs == hm:
        # Maintenant, c'est comme dans la 3.3
            j = 0
            while j < k and s[i+j] == m[j]:
                j += 1
            if j == k:
                return True
        if i < n-k: # Au dernier tour, cela déclencherait une erreur.
            hs = hache_suivant(s, i, hs, k, c)
    return False
```