TP n° 05 bis – Boucles et complexité

I Vérification d'un carré magique

Un carré magique est un arrangement de nombres différents, généralement entiers, dans un tableau dont la somme des éléments sur les lignes, les colonnes et les diagonales sont identiques. On note n la taille des lignes et colonnes, et on parle de carré magique d'ordre n. Le carré magique est dit « normal » s'il est constitué exactement de tous les nombres entiers de 1 à n^2.

(a) Carré magique normal d'ordre 3.

2	7	6
9	5	1
4	3	8

(b) Carré magique d'ordre 3.

23	28	21
22	24	26
27	20	25

(c) Carré magique normal d'ordre 4.

4	14	15	1
9	7	6	12
5	11	10	8
16	2	3	13

TABLE 1 – Exemples de carrés magiques.

1. Vérifier à la main les carrés magiques de la table 1.

Les carrés magiques seront stockés dans des listes en python. Chaque élément de la liste est lui-même une liste contentant une ligne du carré magique de telle sorte que la commande `T[i][j]` permettra d'accéder à l'élément de la ligne i et de colonne j du tableau T. Pour le premier carré magique on a donc : `T = [ [2,7,6] , [9,5,1] , [4,3,8] ]`.
On se propose d'écrire les fonctions permettant de vérifier qu'un tableau passé en argument est bien un carré magique. Si nécessaire, des indications sont proposées en Annexes.

2. Dans un premier temps, la fonction de vérification doit calculer la somme de chaque ligne, de chaque colonne et de chaque diagonale en stockant le résultat dans une liste, puis on effectue la vérification du carré magique. Écrire une fonction `verif_carre_1(T)` qui vérifie si le tableau `T` correspond à un carré magique ou non. La fonction renverra `True` si la carré est magique et `False` sinon.
3. Déterminer la complexité de votre fonction en ne comptant que le nombre d'additions et de comparaisons réalisées dans le meilleur et dans le pire des cas en fonction de n.
4. Cette fonction est-elle optimale ? Proposer une nouvelle fonction `verif_carre_2(T)` pour améliorer les performances et donner la complexité dans le meilleur et dans le pire des cas en fonction de n.

II Calendrier grégorien

La manipulation des dates dans les logiciels de gestion, ou encore sur les sites web de réservation, doit s'effectuer conformément au calendrier grégorien (entré en vigueur à la fin du XVI^{e} siècle en France) en précisant pour chaque date le jour de la semaine. Or le calendrier grégorien est un format assez éloigné des formats habituels de stockage des nombres dans un ordinateur.
On cherche à déterminer, pour une date donnée, sa position dans l'énumération des jours depuis le 1er janvier 1600. et le jour de la semaine correspondant.

1. Proposer une fonction `nombre_de_jours(jours,mois,annee)` qui prend en entrée une date de la forme `jours,mois,annee` postérieure au 1,1,1600 et renvoie le nombre de jours écoulés entre le 1er janvier 1600 et la date considérée, sans tenir compte des années bissextiles (c'est-à-dire en supposant que chaque année comporte 365 jours). On pourra utiliser une liste des nombres de jours de chaque mois pour une année non bissextile : `m = [31,28,31,30,31,30,31,31,30,31,30,31]`.
2. Les années bissextiles sont déterminées par la règle suivante [1] :
   — Si l'année est divisible par 4 et non-divisible par 100, c'est une année bissextile (elle a 366 jours).
   — Si l'année est divisible par 400, c'est une année bissextile (elle a 366 jours).
   — Sinon, l'année n'est pas bissextile (elle a 365 jours).
   Proposer une fonction `bissextile(annee)` renvoyant `True` si l'année est bissextile et `False` sinon.
3. Modifier le programme de la fonction `nombre_de_jours(jours,mois,annee)` pour tenir compte des années bissextiles. Par exemple, `nombre_de_jours(1,2,1600)` doit retourner la valeur 31, `nombre_de_jours(1,1,1604)` la valeur 1461 (366+365+365+365) puisque l'année 1600 est bissextile.
4. Le 1er janvier 2001 était un lundi. Déterminer, en utilisant la fonction de la question précédente, quel jour de la semaine tombera le 1er mai 2040. Quel jour de la semaine est tombé le 14 juillet 1789 ?

1. Wikipédia en français, « Année bissextile », consulté le 23 novembre 2015.

Correction TP n° 05 bis – Boucles et complexité

I Vérification d'un carré magique

2. Code de la fonction de vérification.

```
def verif_carre_1(T):
  n = len(T) #nombre de lignes/colonnes
  ## DETERMINATION DES SOMMES
  resu_somme=[]
  # Boucles sur les lignes
  for i in range(n):
    somme=0
    for j in range(n):
      somme = somme + T[i][j]
    resu_somme.append(somme)
  # Boucles sur les colonnes
  for j in range(n):
    somme=0
    for i in range(n):
      somme = somme + T[i][j]
    resu_somme.append(somme)
  # diagonale 1
  somme=0
  for i in range(n):
    somme = somme + T[i][i]
  resu_somme.append(somme)
  # diagonale 2
  somme=0
  for i in range(n):
    somme = somme + T[i][n-1-i]
  resu_somme.append(somme)

  ##VERIFICATION DU CARRE
  #On verifie que chaque element de la liste est
  #egal au premier element de la liste
  for i in range(1,len(resu_somme)):
    if resu_somme[i] != resu_somme[0]:
      return False # on sort de la fonction en renvoyant False
  # sinon le carre est magique on renvoie True
  return True
```

Listing 1 – Programme naïf.

3. Chaque calcul de somme nécessite n additions[2]. Il faut faire $2n+2$ calculs de somme (n lignes, n colonnes, 2 diagonales).

   Pour les comparaisons, dans le meilleur des cas, la première comparaison conduit à un carré non magique. Dans le meilleur des cas, le nombre d'opérations est donc $(2n+2)n+1 = 2n^2+2n+1$. La complexité est quadratique.

   Dans le pire des cas, c'est-à-dire quand seule la dernière somme est incorrecte ou quand le carré est magique, on réalise $2n+1$ comparaisons. La complexité dans le pire des cas est donc $(2n+2)n+2n+1 = 2n^2+4n+1$ et est donc quadratique également.

2. On pourrait réduire d'une addition en initialisant `somme` à `T[0][0]` au lieu de 0 et en parcourant `range(1,n)`.

4. La fonction n'est pas optimale car on calcule dans tous les cas toutes les sommes avant de faire les comparaisons. Une meilleure approche consisterait à vérifier après chaque calcul de somme si le résultat est identique au résultat précédent ou non. Si ce n'est pas le cas, on sort immédiatement de la fonction, sinon on continue.

   La complexité dans le pire des cas est identique à la fonction précédente et atteinte quand le carré est magique ou que l'erreur est sur la dernière comparaison. La seule différence est une comparaison supplémentaire inutile à la première itération de la première boucle où on compare le premier et le dernier élément d'une liste ne contenant à cette itération qu'un seul élément.

   La complexité dans le meilleur des cas est atteinte lorsque les deux premières sommes sont différentes. On calcule donc deux sommes, soient $2n$ additions, et on fait deux comparaisons. Le nombre d'opération est donc $2n+2$ et la complexité est linéaire Cet algorithme est donc plus performant.

```python
def verif_carre_1(T):
    n = len(T) #nombre de lignes/colonnes
    ## DETERMINATION DES SOMMES
    resu_somme=[]
    # Boucles sur les lignes
    for i in range(n):
        somme=0
        for j in range(n):
            somme = somme + T[i][j]
        resu_somme.append(somme)
        # comparaison du dernier element avec le dernier
        if resu_somme[i] != resu_somme[0]:
            return False # on sort de la fonction en renvoyant False
    # Boucles sur les colonnes
    for j in range(n):
        somme=0
        for i in range(n):
            somme = somme + T[i][j]
        resu_somme.append(somme)
        # comparaison du dernier element avec le dernier
        if resu_somme[i] != resu_somme[0]:
            return False # on sort de la fonction en renvoyant False
    # diagonale 1
    somme=0
    for i in range(n):
        somme = somme + T[i][i]
    resu_somme.append(somme)
    # comparaison du dernier element avec le dernier
    if resu_somme[i] != resu_somme[0]:
        return False # on sort de la fonction en renvoyant False
    # diagonale 2
    somme=0
    # comparaison du dernier element avec le dernier
    if resu_somme[i] != resu_somme[0]:
        return False # on sort de la fonction en renvoyant False
    for i in range(n):
        somme = somme + T[i][n-1-i]
    resu_somme.append(somme)
# sinon le carre est magique, on renvoie True
    return True
```

Listing 2 – Programme optimisé.

II Calendrier grégorien

1. Sans tenir compte des années bissextiles :
   — On compte 365 jours pour toutes les années entièrement écoulées.
   — On ajoute le nombre de jours des mois entièrement écoulés dans l'année en cours.
   — On ajoute le nombre de jours dans le mois en cours.

```
def nombre_de_jours_annees_classiques(jour,mois,annee):
  m = [31,28,31,30,31,30,31,31,30,31,30,31]
  nbjours = (annee-1600)*365 # nombre de jours ecoules pour les annees entieres
  for i in range(mois-1): # mois numerotes de 1 a 12 ; liste indexee de 0 a 11
    nbjours = nbjours + m[i]  # nombre de jours ecoules pour les mois termines
  nbjours = nbjours + jour - 1 # nombre de jours ecoules pour le mois en cours
  return nbjours
```

2. Détermination des années bissextiles.

```
def bissextile(annee):
  if annee % 4 == 0:
    if annee % 100 == 0:
      if annee % 400 == 0:
        bissex = True # annee multiple de 400
      else:
        bissex = False # annee multiple de 4 et de 100 mais pas de 400
    else:
      bissex = True # annee multiple de 4 mais pas de 100
  else:
    bissex = False # annee non multiple de 4
  return bissex
```

3. En tenant compte des années bissextiles :
   — On compte 365 ou 366 jours pour toutes les années entièrement écoulées selon qu'elles sont bissextiles ou non.
   — On ajoute le nombre de jours des mois classiques entièrement écoulés dans l'année en cours. On ajoute un jour si le mois de février de l'année en cours est terminé et que l'année en cours est bissextile.
   — On ajoute le nombre de jours dans le mois en cours.

```
def nombre_de_jours_toutes_annees(jour,mois,annee):
  m = [31,28,31,30,31,30,31,31,30,31,30,31]
  nbjours = 0
  for an in range(1600,annee): # pour tous les ans entierement ecoules
    if bissextile(an):
      nbjours = nbjours + 366 # si annee bissextile on ajoute 366
    else:
      nbjours = nbjours + 365 # sinon on ajoute 365
  for i in range(mois-1):
    nbjours = nbjours + m[i] # nombre de jours ecoules pour les mois termines
  if (mois > 3 or mois == 3) and bissextile(annee):
    nbjours = nbjours + 1 # on ajoute 1 si fevrier est termine et
                          # que l'annee en cours bissextile
  nbjours = nbjours + jour - 1 # nombre de jours ecoules pour le mois en cours
  return nbjours
```

4. Les semaines font toujours 7 jours. On détermine donc uniquement le nombre de jours écoulés depuis le 1er janvier 2001 modulo 7. Si la date testée est postérieure au 1er janvier 2001, le reste est négatif ce qui ne pose pas de souci à python qui gère les indices négatifs dans les listes (-1 étant l'indice du dernier élément, -2 celui de l'avant -dernier, etc.).

```
jours = ["lundi","mardi","mercredi","jeudi","vendredi","samedi","dimanche"]

print(jours[
( nombre_de_jours_toutes_annees(1,5,2040)
- nombre_de_jours_toutes_annees(1,1,2001) ) % 7
])

print(jours[
( nombre_de_jours_toutes_annees(14,7,1789)
- nombre_de_jours_toutes_annees(1,1,2001) ) % 7
])
```