Buller, voter et multiplier

INFORMATIQUE COMMUNE - Devoir n° 3 - Olivier Reynet

Consignes :

1. utiliser une copie différente pour chaque partie du sujet,
2. écrire son nom sur chaque copie,
3. écrire de manière lisible et intelligible,
4. préparer une réponse au brouillon avant de la reporter sur la feuille.

Les trois parties de ce contrôle sont indépendantes.

A Buller

Le tri bulle est un tri comparatif. Son principe est décrit par l'algorithme 1.

Algorithme 1 Tri bulle

```
1 : Fonction FAIRE_MONTER(t, dernier)
2 :     pour j de 0 à dernier-1 répéter
3 :         si t[j] > t[j+1] alors
4 :             ÉCHANGER(t[j], t[j+1])
5 : Fonction TRI_BULLE(t)
6 :     n ← taille(t)
7 :     pour i de n-1 à 1 répéter
8 :         FAIRE_MONTER(t, i)
```

A1. Sur le tableau `[9,5,0,3,1,2,6,4]`, appliquer à la main l'algorithme du tri bulle en détaillant le résultat de chaque itération de boucle de la fonction TRI_BULLE sur le tableau.

Solution :

```
[|9;5;0;3;1;2;6;4|]
[|5;0;3;1;2;6;4;9|]
[|0;3;1;2;5;4;6;9|]
[|0;1;2;3;4;5;6;9|]
[|0;1;2;3;4;5;6;9|]
[|0;1;2;3;4;5;6;9|]
[|0;1;2;3;4;5;6;9|]
[|0;1;2;3;4;5;6;9|]
[|0;1;2;3;4;5;6;9|]
```

A2. Écrire une fonction de prototype `echanger(t,i,j)` qui implémente la fonction ÉCHANGER, c'est-à-dire qui échange l'élément d'indice `i` avec celui d'indice `j` dans une liste `t`.

Solution :

```
def echanger(t, i, j):
    tmp = t[i]
    t[i] = t[j]
    t[j]  = tmp

# ou bien

def echanger(t, i, j):
    t[i], t[j] = t[j], t[i]
```

A3. Écrire une fonction de signature `faire_monter(t,dernier)` qui implémente la fonction homonyme de l'algorithme 1. Celle-ci place l'élément le plus grand du début du tableau à la dernière place possible, c'est-à-dire à l'indice `dernier`.

Solution :

```
    def faire_monter(t,dernier):
        for j in range(dernier):
            if t[j]>t[j+1]:
                echanger(t,j,j+1)
```

A4. Écrire une fonction de signature `tri_bulle(t)` qui implémente le tri bulle d'un tableau d'entiers.

Solution :

```
    def tri_bulle(t):
        for i in range(len(t)-1, 0, -1)
            faire_monter(t,i)
```

R S'il n'y a pas d'échanges lors d'un parcours des éléments qui restent à trier, on peut en conclure que tout le tableau est trié. L'algorithme 2 présente cette optimisation de l'algorithme précédent.

A5. Quelle est la complexité temporelle de la fonction OPT_TRI_BULLE ? On prendra soin de bien préciser de quel(s) paramètre(s) dépend la complexité, de préciser s'il y a un pire ou un meilleur cas et de calculer précisément la complexité.

Solution : La complexité dépend de la taille n du tableau d'entrée.
La complexité dans le meilleur des cas est linéaire : une seule itération de la boucle `while` est nécessaire pour se rendre compte que le tableau est trié. Pour cette itération, la boucle interne

effectue $n-1$ itérations et les instructions des lignes 8, 9 et 10 sont effectuées en un temps constant. D'où la complexité en $O(n)$.

Dans le pire des cas, c'est-à-dire la boucle `while` effectue n itérations, on a :

$$C(n) = \sum_{i=0}^{n} \sum_{j=0} i - 1c = c \sum_{i=0}^{n} i = c\frac{n(n-1)}{2}$$

La complexité est donc en $O(n^2)$.

A6. Écrire une fonction de signature `opt_tri_bulle(t)` qui implémente le tri bulle optimisé.

Solution :

```
1  def opt_tri_bulle(t):
2      trie = False
3      i = len(t) - 1
4      while i > 0 and not trie:
5          trie = True
6          for j in range(i):
7              if t[j] > t[j + 1]:
8                  echanger(t, j, j + 1)
9                  trie = False
10         i -= 1
```

A7. Citer un tri dont la complexité est en $O(n \log n)$ dans tous les cas. Expliquer son principe et illustrer, sur la liste `[9,5,0,3,1,2,6,4]`, le fonctionnement de cet algorithme dans le détail.

Solution : Le tri fusion est en $O(n \log n)$ dans le pire des cas.

`[9 ;5 ;0 ;3 ;1 ;2 ;6 ;4]`
Division en deux tableaux `[9 ;5 ;0 ;3][1 ;2 ;6 ;4]`
Division en deux tableaux `[9 ;5][0 ;3][1 ;2][6 ;4]`
Division en deux tableaux `[9][5][0][3][1][2][6][4]`
Fusion `[5 ;9][0 ;3][1 ;2][4 ;6]`
Fusion `[0 ;3 ;5 ;9][1 ;2 ;4 ;6]`
Fusion `[0 ;1 ;2 ;3 ;4 ;5 ;6 ;9]`

B Voter

La méthode Condorcet[1] est un mode de scrutin, c'est-à-dire une méthode de vote pour faire un choix. Lors d'un tel scrutin, **les électeurs classent tous les candidats dans l'ordre de leur préférence.**

1. Marie Jean Antoine Nicolas de Caritat, marquis de Condorcet, dit Condorcet (1743-1794) était un scientifique brillant, mathématicien, philosophe engagé, homme politique et éditeur français, représentant des Lumières.

Algorithme 2 Tri bulle optimisé

```
1 : Fonction OPT_TRI_BULLE(t)
2 :     n ← taille(t)
3 :     trié ← faux
4 :     i ← n − 1
5 :     tant que t non trié et i > 0 répéter
6 :         trié ← vrai
7 :         pour j de 0 à i-1 répéter
8 :             si t[j] > t[j+1] alors
9 :                 échange(t[j], t[j+1])          ▷ faire monter la bulle
10 :                trié ← faux                    ▷ On n'a donc pas fini de trier...
11 :        i ← i − 1
```

Condorcet a énoncé le principe suivant : «si un choix est préféré à tout autre par une majorité ou une autre, alors ce choix doit être élu.» **Le vainqueur du scrutin, s'il existe, est le candidat qui bat tous les autres en duel.** Un tel candidat est appelé **vainqueur de Condorcet**.

■ **Exemple 1 — Scrutin de Condorcet.** Soit trois candidats A,B, et C à une élection. On note $\succ$ la relation «est préféré à». Ainsi $A \succ B$ signifie que A est préféré à B. Le bulletin type d'un électeur pour ce scrutin est donc : $A \succ C \succ B$.

Lors du dépouillement, on a compté :

- 23 votants : $A \succ C \succ B$;
- 19 votants : $B \succ C \succ A$;
- 16 votants : $C \succ B \succ A$;
- 2 votants : $C \succ A \succ B$.

Lorsqu'on compare deux à deux tous les choix, on obtient :

$\succ$	A	B	C
A	0	25	23
B	35	0	19
C	37	41	0

On en déduit que la préférence majoritaire est $C \succ B \succ A$ et que C est le choix préféré à tous les autres.

On peut construire à partir de ce tableau la **matrice des duels** M. Dans cette matrice, chaque terme m_{ij} indique le nombre de votes en faveur de la préférence $i \succ j$. Par ailleurs, A a pour indice 0, B a pour indice 1 et C 2.

$$M = \begin{pmatrix} 0 & 25 & 23 \\ 35 & 0 & 19 \\ 37 & 41 & 0 \end{pmatrix}$$

Dans l'optique d'automatiser le scrutin Condorcet, on représente :

- chaque candidat par un entier distinct, en commençant à zéro. Pour trois candidats, on aura donc les candidats 0,1 et 2.
- un bulletin de vote d'un électeur par une liste contenant les candidats dans l'ordre de préférence de l'électeur,

- un scrutin est une liste de bulletins,
- une matrice de duels est une liste de listes.

B1. Écrire une fonction de signature `matrice_nulle(n)` qui renvoie une matrice de taille (n, n) ne contenant que des zéros sous la forme d'une liste de liste.

Solution :

```
def matrice_nulle(n):
    M = []
    for i in range(n):
        L = []
        for j in range(n):
            L.append(0)
        M.append(L)
    return M
    #return  [[0 for _ in range(n)] for _ in range(n)]
```

■ **Exemple 2 — Comptabiliser un bulletin.** On cherche à effectuer le dépouillement, c'est-à-dire à comptabiliser les votes de chaque électeur.

On suppose qu'il y a trois candidats 0, 1 et 2. Soit `b = [2,0,1]` un bullletin.
Soit `M = [[0, 25, 23], [35, 0, 19], [37, 41, 0]]` la matrice des duels déjà partiellement construite.
Lorsqu'on aura comptabilisé ce bulletin, la matrice `M` vaudra :
`[[0, 26, 23], [35, 0, 19], [38, 42, 0]]`.

B2. Écrire une fonction de signature `depouiller_un_bulletin(bulletin, M)` où `bulletin` est un bulletin d'électeur et `M` la matrice des duels à compléter. Cette fonction complète la matrice des duels d'après le choix de l'électeur.

Solution :

```
def depouiller_un_bulletin(bulletin, M):
    n = len(M)
    for i in range(n-1):
        for j in range(i + 1, n):
            M[bulletin[i]][bulletin[j]] += 1
```

B3. Écrire une fonction de signature `depouiller(bulletins)` où `bulletins` est un scrutin, c'est-à-dire la liste de tous les bulletins des électeurs. Cette fonction renvoie la matrice des duels `M` complétée de ce scrutin.

Solution :

```
def depouiller(bulletins):
    assert len(bulletins) > 0
    n = len(bulletins[0])
    M = matrice_nulle(n)
    for bulletin in bulletins:
        depouiller_un_bulletin(bulletin, M)
```

```
    return M
```

■ **Exemple 3 — Duels gagnés.** Lorsqu'on cherche à déterminer le vainqueur du scrutin de Condorcet, on a besoin de savoir le nombre de duels que chaque candidat a remporté. Soit M la matrice des duels. On sait que i gagné son duel avec $j \neq i$ si $m_{ij} > m_{ji}$. Le nombre de duels gagnés par i est donc incrémenté à chaque fois que cette inégalité est vraie.

B4. Écrire une fonction de signature `nb_duels_gagnes(M, i)` où `M` est la matrice des duels et `i` un numéro de candidat. Cette fonction renvoie le nombre de duels gagnés par le candidat `i`.

Solution :

```
def nb_duels_gagnes(M, i):
    nb_duels = 0
    for j in range(n):
        if i != j and M[i][j] > M[j][i]:
            nb_duels += 1
    return nb_duels
```

B5. Dans un scrutin de Condorcet, on a décompté :

- 41 bulletins `[0, 1, 2]`
- 33 bulletins `[1, 2, 0]`
- 22 bulletins `[2, 0, 1]`

À quelle condition un candidat a-t-il gagné ? Y-a-t-il un vainqueur de Condorcet ?

Solution : Un candidat a gagné s'il gagne tous ses duels, soit 2 duels. Les candidats n'ont gagné qu'un seul duel chacun. Il n'y a donc pas de vainqueur de Condorcet.

B6. Écrire une fonction de signature `determiner_vainqueur(M)` où `M` est la matrice des duels d'un scrutin de Condorcet. Cette fonction renvoie le vainqueur de Condorcet s'il existe et `None` sinon.

Solution :

```
def determiner_vainqueur(M):
    assert len(M) > 0
    n = len(M)
    for i in range(n):
        if nb_duels_gagnes(M, i) == n-1:
            return i
    return None
```

C Multiplier

L'algorithme de Karatsuba permet de multiplier rapidement deux nombres de n chiffres, plus rapidement qu'avec la méthode naïve que l'on utilise à la main. Dans cette partie, on se propose de le programmer.

C1. Soit a et b deux nombres de n chiffres. Combien de multiplications et combien d'additions sont nécessaires pour calculer le produit $a \times b$ avec la méthode naïve (à la main) ?

Solution : Chaque chiffre de a est multiplié par chaque chiffre de b. On fait $n \times n$ multiplications et n additions.

C2. Quelle est la complexité de cet algorithme ?

Solution : La complexité est liée au nombre de multiplications et d'additions à un seul chiffre nécessaires et donc directement liée au nombre n de chiffres mis en jeu dans l'opération. Donc en $O(n^2 + n) = O(n^2)$. Les additions sont négligeables devant les multiplications.

C3. Écrire une fonction de signature `chiffres_b10(a : int)-> int` où `a` est un nombre entier positif à n chiffres en base 10. Cette fonction renvoie n et procède de manière itérative avec une boucle `while`.

Solution :

```
def chiffres_b10(a):
    assert a >= 0
    count = 1
    while a >= 10:
        a = a // 10
        count += 1
    return count
```

C4. Prouver que la fonction `chiffres_b10` termine.

Solution : `a` est une quantité entière positive qui décroît strictement à chaque itération puisqu'elle est divisée par dix. Les divisions euclidiennes successives feront donc tomber `a` à zéro et invalideront la condition de la boucle `while`. La fonction termine.

C5. En vous appuyant sur la division entière, écrire une fonction de signature `r_chiffres_b10(a : int)-> int`. Cette fonction renvoie le même résultat que la fonction `chiffres_b10` mais procède de manière **récursive**.

Solution :

```
def r_chiffres_b10(a):
    assert a >= 0
    if a // 10 == 0:
        return 1
```

```
    else:
        return 1 + r_chiffres_b10(a // 10)
```

C6. Quelle est la complextié de la fonction `r_chiffres_b10` ?

Solution : La complexité dépend du nombre n de chiffres du nombre a.
La récurrence de la complexité de la fonction s'écrit : $T(n) = c + T(n-1)$ puisqu'en dehors de l'appel récursif la fonction n'exécute que des opérations de complexité constante $O(1)$ et que l'appel récursif s'effectue sur un nombre possédant un chiffre de moins en base 10 que le nombre a.
C'est une suite arithmétique de raison c et de premier terme $T(0)$ qui vaut c également, car la condition d'arrêt s'exécute en un temps constant. C'est pourquoi, $T(n) = cn+c = c(n+1) = O(n)$.
La complexité de `r_chiffres_b10` est linéraire par rapport au nombre de chiffre de a.

L'idée à la base de l'algorithme 3 de Karatsuba est l'observation suivante. Dans le calcul qui suit, le développement 2 ne nécessite que trois multiplications au lieu de quatre pour le développement 1 :

$$\left(a \times 10^k + b\right)\left(c \times 10^k + d\right) = ac \times 10^{2k} + (ad + bc) \times 10^k + bd \quad (1)$$

$$= ac \times 10^{2k} + (ac + bd - (a-b)(c-d)) \times 10^k + bd \quad (2)$$

On réutilise en effet deux fois ac et bd.

Considérons deux nombres à n chiffres a et b. En prenant $k = \frac{n}{2}$ et en effectuant la division euclidienne avec 10^k sur chaque nombre, la méthode peut être appliquée de manière récursive pour les calculs de ac, bd et $(a-b)(c-d)$ en scindant à nouveau a, b, c et d en deux et ainsi de suite.

Algorithme 3 Karatsuba

```
1 : Fonction KARATSUBA(x, y)
2 :     si x < 10 ou y < 10 alors
3 :         renvoyer x × y                    ▷ Condition d'arrêt : un seul chiffre
4 :     n ← le nombre de chiffres maximum en base 10 de x et y
5 :     k ← n//2
6 :     a, b ← DIVMOD(x, 10^k)
7 :     c, d ← DIVMOD(y, 10^k)
8 :     ac ← KARATSUBA(a, c)
9 :     bd ← KARATSUBA(b, d)
10 :    ad_bc ← KARATSUBA(a + b, c + d) − ac − bd
11 :    renvoyer ac × 10^(2k) + ad_bc × 10^k + bd
```

C7. Écrire une fonction de signature `karatsuba(x, y)` qui implémente l'algorithme 3. L'usage de la fonction `divmod(a,b)` qui renvoie le quotient (`a//b`) et le reste (`a%b`) de la division euclidienne de `a` par `b` est autorisé.

Solution :

```python
def karatsuba(x, y):
    if x < 10 or y < 10:
        return x * y  # condition d'arrêt

    # Taille des nombres
    n = vmax(chiffres_b10(x), chiffres_b10(y))
    k = n // 2

    # Division du pb en deux
    a, b = divmod(x,10 ** k)   # O(n)
    c, d = divmod(y,10 ** k)   # O(n)

    # Appels récursifs
    ac = karatsuba(a, c)
    bd = karatsuba(b, d)
    ad_bc = karatsuba((a + b), (c + d)) - ac - bd

    # Combinaison des résultats
    return ac * 10 ** (2 * k) + ad_bc * 10 ** k + bd
```

C8. À quelle grande famille d'algorithmes appartient l'algorithme de Karatsuba? Citer un autre algorithme de cette famille dont vous donnerez la complexité.

Solution : C'est un algorithme de type diviser pour régner avec $d = 2$ et $r = 3$, c'est-à-dire divisions par deux de la taille du problème et trois appels récursifs. Le tri fusion en $O(n \log n)$ fait parti de cette famille.

C9. Proposer une relation de récurrence qui caractérise la complexité de la fonction `karatsuba`. On précisera quel est le paramètre agissant sur la complexité. L'opération `divmod(a,b)` a une complexité constante par rapport au nombre de chiffres de `a` et de `b`.

Solution : Soit n le nombre de chiffres de x et y.
$T(n) = c + 2n + 3T(n//2)$
Les opérations de multiplication et d'addition s'effectuent en un temps constant. Les deux divisions euclidiennes sont linéaires par rapport à n. On procède à trois appels récursifs à chaque appel de la fonction sur des nombres dont le nombre de chiffre a été divisé par deux.

C10. On peut montrer que la complexité de `karatsuba` $T(n)$ est telle que : $T(n) \leqslant 3T(n/2)$. En supposant que $n = 2^k$, c'est-à-dire n est une puissance de 2, et en introduisant une suite auxiliaire, montrer que : $T(n) = O(n^{\log_2 3})$. Est-ce mieux que l'algorithme naïf[2] ?

Solution : On peut toujours majorer un entier par une puissance de deux. Cette approche est donc compatible avec l'étude asymptotique. On pose $n = 2^k$.

2. $\log_2 3 \approx 1.585$

$T(2^k) \leqslant 3T(2^{k-1})$.
En considérant l'égalité et en introduisant une suite $v_k = T(2^k)$, on peut écrire que $v_k = 3v_{k-1}$. L'inégalité nous dit que v_k ne croit pas plus vite que $3v_{k-1}$, l'égalité étant le maximum de sa croissance. v est une suite géométrique de raison 3 et de premier terme $v_0 = T(1) = c$, une constante correspondant à l'exécution de la condition d'arrêt de l'algorithme. Donc, $v_k = c3^k$.
On en déduit que $T(n) \leqslant c3^{\log_2 n} = c(2^{\log_2 3})^{\log_2 n} = c(2^{\log_2 n})^{\log_2 3} = cn^{\log_2 3} = O(n^{\log_2 3}) = O(n^{1.585})$.
C'est mieux que l'algorithme naïf car $1.585 < 2$.