Algorithmique

Tri à bulles de la complexité à la preuve

B2G3A

Jean Dominique CECCALDI - Philippe CLOUP -Marc ELDIN - Luc VINCENT

Le programme

Contenus	Capacités attendues	Commentaires
Tris par insertion, par sélection	Écrire un algorithme de tri. Décrire un invariant de boucle qui prouve la correction des tris par insertion, par sélection.	La terminaison de ces algorithmes est à justifier. On montre que leur coût est quadratique dans le pire cas.

Les prérequis

On suppose que les élèves ne sont pas débutants avec Python et qu'ils maitrisent les boucles et les tests.

Objectifs

On se propose de leur faire découvrir un algorithme de tri à bulles et d'en étudier la complexité. Les élèves travaillent en autonomie ou par petits groupes de 2 ou 3 en suivant la fiche élève qu'ils complètent.

Il y a aussi dans toute l'activité, de façon sous-jacente, les réflexes à avoir pour tester et debugger un programme.

Durée

4 Heures

FICHE ELEVE

(Version professeur, commentée en rouge)

Découverte de la fonction bulle(a)

1. Recopier le programme suivant et l'enregistrer.

   On donne ici le programme sous python d'une fonction bulle(a) qui prend comme argument un tableau et qui en renvoie un autre.

```
1  def bulle(a):
2      n=len(a)
3      for i in range(n-1):
4          for j in range (n-1,i,-1):
5              if a[j]<a[j-1]:
6                  a[j],a[j-1]=a[j-1],a[j]
7      return a
```

2. Quelle est la nature de la variable a ?
   a est une variable de type tableau (liste en python).

3. On suppose que la variable « **a** » est un tableau à 1 élément. Que peut-on conjecturer du résultat de la fonction bulle(a) ?
   Elle ne change rien.

4. On suppose que la variable « **a** » est un tableau dont les éléments sont ordonnés de façon croissante. Que peut-on conjecturer du résultat de la fonction bulle(a) ?
   Elle ne change rien.

5. A l'aide d'exemples de tableau « **a** » quelconques de votre choix, émettre une conjecture sur la fonction bulle(a). Cela est-il compatible avec les points 3 et 4 ?
   Il semble que bulle(a) trie le tableau par ordre croissant.

Analyse la fonction bulle(a)

1. Ajouter l'instruction « print("i: ", i," j: ", j," a: ", a) » dans le programme afin d'afficher les différentes étapes de la fonction bulle(a) et coller les résultats avec le tableau a = [41, 33, 27, 22, 17, 9, 5].
   On ne dit pas aux élèves où rajouter l'instruction, c'est à eux de trouver le choix le plus pertinent.

```
1  def bulle(a):
2      n=len(a)
3      for i in range(n-1):
4          for j in range (n-1,i,-1):
5              if a[j]<a[j-1]:
6                  a[j],a[j-1]=a[j-1],a[j]
7                  print("i: ",i," j: ",j," a: ",a)
8      return a
```

Résultats obtenus

i: 0 j: 6 a: [41, 33, 27, 22, 17, 5, 9]

i: 0 j: 5 a: [41, 33, 27, 22, 5, 17, 9]

i: 0 j: 4 a: [41, 33, 27, 5, 22, 17, 9]

i: 0 j: 3 a: [41, 33, 5, 27, 22, 17, 9]

i: 0 j: 2 a: [41, 5, 33, 27, 22, 17, 9]

i: 0 j: 1 a: [5, 41, 33, 27, 22, 17, 9]

i: 1 j: 6 a: [5, 41, 33, 27, 22, 9, 17]

i: 1 j: 5 a: [5, 41, 33, 27, 9, 22, 17]

i: 1 j: 4 a: [5, 41, 33, 9, 27, 22, 17]

i: 1 j: 3 a: [5, 41, 9, 33, 27, 22, 17]

i: 1 j: 2 a: [5, 9, 41, 33, 27, 22, 17]

i: 2 j: 6 a: [5, 9, 41, 33, 27, 17, 22]

i: 2 j: 5 a: [5, 9, 41, 33, 17, 27, 22]

i: 2 j: 4 a: [5, 9, 41, 17, 33, 27, 22]

i: 2 j: 3 a: [5, 9, 17, 41, 33, 27, 22]

i: 3 j: 6 a: [5, 9, 17, 41, 33, 22, 27]

i: 3 j: 5 a: [5, 9, 17, 41, 22, 33, 27]

i: 3 j: 4 a: [5, 9, 17, 22, 41, 33, 27]

i: 4 j: 6 a: [5, 9, 17, 22, 41, 27, 33]

i: 4 j: 5 a: [5, 9, 17, 22, 27, 41, 33]

i: 5 j: 6 a: [5, 9, 17, 22, 27, 33, 41]

2. Décrire la « progression » d'un élément du tableau en fonction des différentes étapes de la fonction bulle(a).
   Il faut constater qu'à chaque série de j, le plus petit élément remonte vers la surface (comme une bulle) de la droite vers la gauche par permutation avec l'un de ses voisins.
3. Que pensez-vous du tableau choisi ? Le cas est il le plus favorable, « moyen » ou pire ?
   C'est le pire cas.
4. Vérifier le fonctionnement sur un tableau déjà trié par ordre croissant.

Complexité de l'algorithme de la fonction bulle(a)

Nous allons essayer de nous intéresser à la complexité de l'algorithme et en particulier nous allons essayer de compter le nombre de permutions de cases du tableau dans le pire des cas.

1. On choisit comme indicateur de complexité de compter le nombre de permutations à chaque utilisation de la fonction bulle(a). Modifier le programme pour cela.

```
def bulle(a):
    n=len(a)
    nb_etapes=0
    for i in range(n-1):
        for j in range (n-1,i,-1):
            if a[j]<a[j-1]:
                a[j],a[j-1]=a[j-1],a[j]
                nb_etapes+=1
    return (a, nb_etapes)
```

2. Quels sont les tableaux qui nécessitent le plus de permutations ?
   Les tableaux qui sont classés en ordre inverse représentent le pire des cas à trier.
3. Combien y-a-t-il de permutations avec les tableaux suivants :
   [41,33,27,22,17,9,5], [41,33,27,22,17,9], [41,33,27,22,17], [41,33,27,22], [41,33,27], [41,33], [41] ?
   Exprimer vos résultats sous la forme d'une somme et placez les résultats dans le tableau suivant :

Longueur n du tableau	1	2	3	4	5	6	7
Nombre de permutations	0	1	3	6	10	15	21

On attend des élèves qu'ils comprennent que le « pire » des cas est celui où le tableau est trié par ordre décroissant…

On constate ici que dans le « pire » des cas, il faut 21 étapes pour un tableau de 7 éléments (6x3/2) ; les élèves doivent pouvoir exprimer ce résultat sous la forme 6+5+4+3+2+1.

4. Construire un graphe avec en abscisse le nombre d'éléments du « pire » tableau et en ordonnée le nombre de permutations que le tri réclame. Reconnaissez-vous la courbe ?

Ils reconnaitront peut-être une parabole.

5. Le site http://oeis.org vous permet d'obtenir la fonction et de tracer la courbe sur des tableaux de grande taille.
   Attendu :

A000217 Triangular numbers: a(n) = binomial(n+1,2) = n(n+1)/2 = 0 + 1 + 2 + ... + n. +30 3679
(Formerly M2535 N1002)

0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 120, 136, 153, 171, 190,
210, 231, 253, 276, 300, 325, 351, 378, 406, 435, 465, 496, 528, 561, 595, 630, 666,
703, 741, 780, 820, 861, 903, 946, 990, 1035, 1081, 1128, 1176, 1225, 1275, 1326, 1378,
1431 (list; graph; refs; listen; history; text; internal format)

On peut admettre que lorsque n devient très grand (n tend vers plus l'infini) 1 est négligeable devant n. Donc a(n) a pour approximation n^2.

On admet que la courbe a pour équation $f(x) = \frac{1}{2}(x^2-x)$. Cela vous parait-il cohérent avec le résultat du site ?

Ils connaissent peut-être la formule : 1+2+3+… ; Ils peuvent tester la formule sur les valeurs du tableau (conjecture) ; ils peuvent résoudre un système de 3 équations à 3 inconnues (à la main ou à l'aide d'un logiciel de calcul formel du type Wolfram Alpha).

Dans cette question, nous avons voulu faire comprendre aux élèves la notion de complexité d'un algorithme : dans le cas présent, en se plaçant toujours dans le pire des cas, la complexité est d'ordre « n^2 ». En effet le x est négligeable devant le x^2.

Pertinence de l'indicateur de complexité

Dans l'étude précédente nous avons considéré le pire des cas, une liste triée en ordre décroissant. Observons maintenant notre indicateur de complexité sur les meilleurs cas.

1. Quels sont les tableaux qui nécessitent le moins de permutations ?
   Les tableaux qui sont classés en ordre croissant représentent le meilleur des cas à trier.

2. Combien y-a-t-il de permutations avec les tableaux suivants :
   [5,9,17,22,27,33,41], [9,17,22,27,33,41], [17,22,27,33,41], [22,27,33,41], [27,33,41], [33,41], [41] ?
   Exprimer vos résultats sous la forme d'une somme et placer les résultats dans le tableau suivant :

Longueur n du tableau	1	2	3	4	5	6	7
Nombre de permutations	0	0	0	0	0	0	0

3. Mesure du temps d'exécution sur les tableaux triés précédents à l'aide du programme suivant
   Il faut importer le module « time » et on peut demander aux élèves de faire une recherche sur la façon de mesurer le temps d'exécution d'un programme.

```
import time

def bulle(a):
    n=len(a)
    nb_etapes=0
    for i in range(n-1):
        for j in range (n-1,i,-1):
            if a[j]<a[j-1]:
                a[j],a[j-1]=a[j-1],a[j]
                nb_etapes+=1
    return (a, nb_etapes)

#test de la fonction
t_ini=time.time()
nb_testMax=100000
for n_test in range(nb_testMax):
    c=[5,9,17,22,27,33,41]
    bulle(c)
t_fin=time.time()
print((t_fin - t_ini)/nb_testMax)
```

Ordre de grandeur pour le tableau [5,9,17,22,27,33,41] : 5.5 e-06 s

4. Que peut-on conclure ?
   Le nombre de permutation est constant à 0 quel que soit le nombre d'élément dans le tableau. La complexité devrait varier. L'indicateur nombre de permutation n'est donc pas pertinent dans un cas autre que le pire des cas.

5. Modifier le programme de la façon suivante :

```
def bulle(a):
    n=len(a)
    nb_etapes=0
    for i in range(n-1):
        for j in range (n-1,i,-1):
            nb_etapes+=1
            if a[j]<a[j-1]:
                a[j],a[j-1]=a[j-1],a[j]
    return (a, nb_etapes)
```

6. Quelle est la valeur de nb_etapes maintenant, pour les tableaux suivants : [5,9,17,22,27,33,41], [9,17,22,27,33,41], [17,22,27,33,41], [22,27,33,41], [27,33,41], [33,41], [41] ?

Longueur n du tableau	1	2	3	4	5	6	7
nb_etapes	0	1	3	6	10	15	21

7. Que compte maintenant la variable nb_etapes ?
   Le nombre de comparaison alors que nous comptions le nombre de permutations

Dans le cas présent, en se plaçant dans le pire des cas, ou dans le meilleur des cas, ou dans n'importe quel cas, la complexité reste d'ordre « n^2 ».

Dans cette question, nous avons voulu faire comprendre aux élèves que le choix de l'indicateur nb_etapes est arbitraire mais doit refléter une tendance réelle du couple (Algorithme, Données).

Lien entre temps d'exécution et dimension du tableau

En travaillant un peu à la louche, on peut dire que le temps d'exécution d'un programme est fonction de la taille du tableau ; en particulier que si on triple le nombre de cases du tableau, le temps est multiplié par 9.

Essayons de le vérifier avec Python.

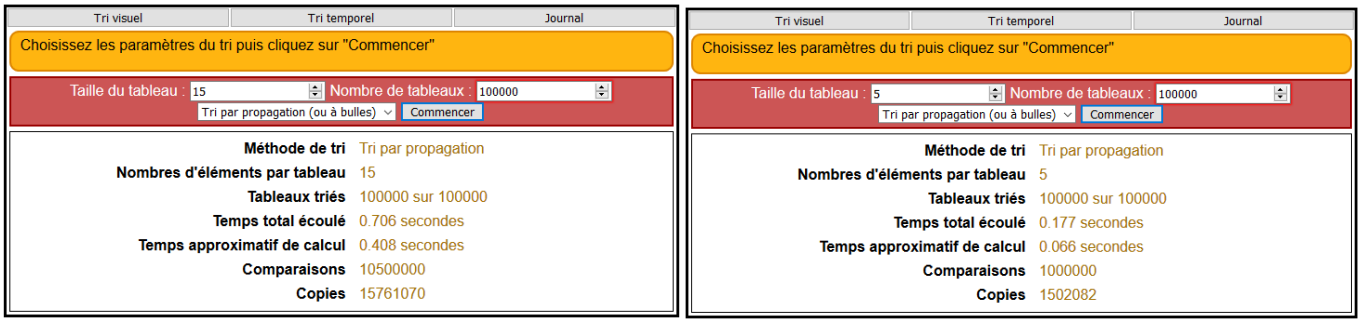
```
1  import time
2
3  def bulle(a):
4      n=len(a)
5      for i in range(n-1):
6          for j in range (n-1,i,-1):
7              if a[j]<a[j-1]:
8                  a[j],a[j-1]=a[j-1],a[j]
9      return a
10
11 #test de la fonction
12 t_ini=time.time()
13 for n_test in range(100000):
14     l=[150,140,130,100,99,90,80,70,41,33,27,22,17,5]
15     bulle(l)
16 t_fin=time.time()
17 print(t_fin - t_ini)
```

1. Combien de fois est exécutée le tri sur le tableau l de 15 éléments ?
   ligne 13 on réalise 100000 tri car la machine renvoie 0 s pour un tri unique.
2. *Modifier le programme pour effectuer le test sur un tableau c=[27,22,17,9,5] et compléter le tableau*

Tableau	l	c
Longueur n	15	5
Durée ms	3.798	0.546

Une multiplication par 3 du nombre d'élément a multiplié le temps par 7 ...

3. Comparer avec le résultat obtenu sur le site https://interstices.info/les-algorithmes-de-tri/



On retrouve ici un rapport proche de 7 la valeur théorique 9 (complexité en n^2) est une limite.

Terminaison

Comment faire la preuve qu'un programme se termine ?

On modifie le programme précédent de la façon suivante :

```
1  def bulle(a):
2      n=len(a)
3      for i in range(n-1):
4          for j in range (n-1,i,-1):
5              if a[j]<a[j-1]:
6                  a[j],a[j-1]=a[j-1],a[j]
7          print("i: ",i," a: ",a)
8      return a
```

On observe deux boucles « **for** » imbriquées de taille inférieure ou égale à n.

Boucle interne : lignes 4-6

Boucle principale : lignes 3-7

Règle : Si dans une boucle for l'indice n'est pas modifié, il n'y a pas de problème de terminaison.

1. Justifier que le programme se termine

Boucle interne : indicée par j qui n'est pas modifié.

Boucle principale : Indicée par i qui n'est pas modifié.

Invariant de boucle

Pour faire la preuve du programme, on cherche à identifier un **invariant de boucle**.

On reprend le programme précédent :

```
1  def bulle(a):
2      n=len(a)
3      for i in range(n-1):
4          for j in range (n-1,i,-1):
5              if a[j]<a[j-1]:
6                  a[j],a[j-1]=a[j-1],a[j]
7          print("i: ",i," a: ",a)
8      return a
```

1. Tester le programme sur le tableau [100,11,25,8,12,4,17,5,9,1,6,7]

```
>>> bulle([100,11,25,8,12,4,17,5,9,1,6,7])
i:  0  a:  [1, 100, 11, 25, 8, 12, 4, 17, 5, 9, 6, 7]
i:  1  a:  [1, 4, 100, 11, 25, 8, 12, 5, 17, 6, 9, 7]
i:  2  a:  [1, 4, 5, 100, 11, 25, 8, 12, 6, 17, 7, 9]
i:  3  a:  [1, 4, 5, 6, 100, 11, 25, 8, 12, 7, 17, 9]
i:  4  a:  [1, 4, 5, 6, 7, 100, 11, 25, 8, 12, 9, 17]
i:  5  a:  [1, 4, 5, 6, 7, 8, 100, 11, 25, 9, 12, 17]
i:  6  a:  [1, 4, 5, 6, 7, 8, 9, 100, 11, 25, 12, 17]
i:  7  a:  [1, 4, 5, 6, 7, 8, 9, 11, 100, 12, 25, 17]
i:  8  a:  [1, 4, 5, 6, 7, 8, 9, 11, 12, 100, 17, 25]
i:  9  a:  [1, 4, 5, 6, 7, 8, 9, 11, 12, 17, 100, 25]
i:  10  a:  [1, 4, 5, 6, 7, 8, 9, 11, 12, 17, 25, 100]
```

2. A la fin de l'itération i=0 combien d'élément ont été triés ? *1*
3. A la fin de l'itération i=6 combien d'élément ont été triés et où se trouvent-ils ? *7 en partant de la gauche.*
4. Quel est le rôle du bloc encadré en vert ?
   La boucle interne va placer le minimum trouvé à gauche de la partie non triée.
5. Enoncer une propriété du tableau vrai pour chaque itération de tri k

Après k itérations, k+1 éléments sont classés par ordre croissant à gauche du tableau. Cette propriété est appelée ***invariant de boucle.***

Préconditions postconditions

Préconditions : On définit le format des données entrantes. *Un tableau d'entiers de dimension n.*

Postconditions : On définit le format des données sortantes. *Un tableau classé d'entiers de dimension n.*

Preuve

Pour vérifier la correction de cet algorithme on vérifie que si l'entrée satisfait les préconditions alors les postconditions sont respectées. Pour on réalise la preuve de l'algorithme.

Cas de base : initialisation k=0 le plus petit élément du tableau initial est classé.

Hérédité : On suppose que le tableau est trié jusqu'à la position k, on cherche dans le reste du tableau le minimum, qu'on permute à la position k+1.

Travail complémentaire (Optimisation du programme)

Tester le programme sur le tableau [10,11,25,8,12,4,17,5,9,1,6,7]

```
i:  0  a:  [1, 10, 11, 25, 8, 12, 4, 17, 5, 9, 6, 7]
i:  1  a:  [1, 4, 10, 11, 25, 8, 12, 5, 17, 6, 9, 7]
i:  2  a:  [1, 4, 5, 10, 11, 25, 8, 12, 6, 17, 7, 9]
i:  3  a:  [1, 4, 5, 6, 10, 11, 25, 8, 12, 7, 17, 9]
i:  4  a:  [1, 4, 5, 6, 7, 10, 11, 25, 8, 12, 9, 17]
i:  5  a:  [1, 4, 5, 6, 7, 8, 10, 11, 25, 9, 12, 17]
i:  6  a:  [1, 4, 5, 6, 7, 8, 9, 10, 11, 25, 12, 17]
i:  7  a:  [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 25, 17]
i:  8  a:  [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 17, 25]
i:  9  a:  [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 17, 25]
i:  10  a:  [1, 4, 5, 6, 7, 8, 9, 10, 11, 12, 17, 25]
```

Que constatez-vous ?

L'algorithme continue à trier un tableau déjà trié : les deux derniers tours (i = 9 et i = 10) sont inutiles.

Programme optimisé

On donne le programme suivant

```
1  def bulle(a):
2      n=len(a)
3      trie=False
4      while not trie:
5          trie=True
6          for j in range (n-1,0,-1):
7              if a[j]<a[j-1]:
8                  a[j],a[j-1]=a[j-1],a[j]
9                  trie =False
10     return a
```

Tester le programme sur le tableau [10,11,25,8,12,4,17,5,80,99,100,125]. Les instructions permettant d'afficher le nombre d'itérations ont été masquées.

```
k:  1  a:  [4, 10, 11, 25, 8, 12, 5, 17, 80, 99, 100, 125]
k:  2  a:  [4, 5, 10, 11, 25, 8, 12, 17, 80, 99, 100, 125]
k:  3  a:  [4, 5, 8, 10, 11, 25, 12, 17, 80, 99, 100, 125]
k:  4  a:  [4, 5, 8, 10, 11, 12, 25, 17, 80, 99, 100, 125]
k:  5  a:  [4, 5, 8, 10, 11, 12, 17, 25, 80, 99, 100, 125]
```

Qu'apporte ce nouvel algorithme ? *Il nécessite moins d'itérations*

Quelles sont les propriétés que nous devrions connaître sur ce nouvel algorithme de tri afin de la valider ?