Université de Lille 1 SCIENCES ET TECHNOLOGIES

ufr d'IEEA
Formations en
Informatique de
Lille 1

L1 MASS/SESI/PEIP - 2014–2015

Algorithmes et Programmation 1

Devoir surveillé 2

Mardi 12 mai 2015 - durée 2h00 - documents et calculatrices non autorisés

Veuillez indiquer le numéro de votre groupe de TD sur la copie qu'il est inutile de rendre anonyme.
Ce sujet contient trois exercices indépendants.
Toute fonction que vous réaliserez devra être documentée et les contraintes d'utilisation précisées.

Exercice 1 *Boules de cristal*

Dans cet exercice, on se propose d'étudier des algorithmes permettant de déterminer l'*étage critique* d'un immeuble. On appelle *étage critique* d'un immeuble l'étage à partir duquel on est sûr qu'une boule de cristal lachée de cet étage se brise en atteignant le sol. Au dessous de cet étage critique, on est sûr que si on lache une boule de cristal alors elle reste intacte. On considère que le monde est idéal, si on renouvelle l'expérience, on obtient toujours le même résultat.

On suppose définie la variable globale nommée `IMMEUBLE` dont la valeur est un dictionnaire avec deux clés :

- l'une nommée, `'nb_etages'` qui contient le nombre d'étages. C'est un entier naturel non nul. Les étages sont numérotés à partir de `0` jusqu'à `immeuble['nb_etages']-1`,
- l'autre nommée `'casse'` dont la valeur associée est un prédicat qui indique si une boule lancée du i-ème étage casse ou non. Par exemple la session qui suit indique que lancée du 4ème étage une boule de cristal ne casse pas en atterissant, tandis que lancée du 12ème étage elle se brise.

```
>>> IMMEUBLE['casse'](4)
False
>>> IMMEUBLE['casse'](12)
True
```

Question 1 En supposant qu'on ne dispose que d'une seule boule de cristal, expliquez comment procéder pour déterminer l'étage critique?

Dans la suite, on suppose que le nombre initial de boules de cristal est au moins égal à 2.
La fonction suivante implante un algorithme permettant de déterminer l'étage critique :

```
1  def algo_racine ():
2      """
3      NoneType -> int
4      renvoie le numéro du premier étage où cela casse
5
6      CU : le nombre d'étages de l'immeuble est un carré parfait
7      """
8      grand_pas = int(sqrt(IMMEUBLE['nb_etages'])) # racine carrée du nbre d'étages
9      assert IMMEUBLE['nb_etages'] == grand_pas ** 2
10     i = 0
11     while i < IMMEUBLE['nb_etages'] and not IMMEUBLE['casse'](i) :
12         i = i + grand_pas
13     if i == 0 :
14         return i
```

```
15      else:
16          j = i - grand_pas + 1
17          while j < i and not IMMEUBLE['casse'](j) :
18              j = j + 1
19          return j
```

La fonction `sqrt` apparaissant à la ligne 8 est celle du module `math` calculant la racine carrée d'un nombre. On la suppose préalablement importée.

Question 2 En supposant dans cette question que l'immeuble possède neuf étages (`IMMEUBLE['nb_etages'] = 9`) et que les boules cassent à partir de l'étage 5, présentez sous forme d'un tableau les valeurs prises par les variables `grand_pas`, `i` et `j` et la valeur renvoyée lors de l'appel `algo_racine()`.

Question 3 En fonction de `nb_etages`, déterminez le meilleur des cas. Dans ce cas combien de lâchers de boules (appels à `IMMEUBLE['casse']`) sont effectués.

Question 4 Même question pour le pire des cas.

Question 5 On suppose dans cette question que le nombre de boules disponibles est au moins égal à 2. On propose l'idée d'algorithme suivant :

— tant qu'on dispose d'au moins deux boules on fait une recherche dichotomique

— avec la dernière boule on termine par une recherche séquentielle.

Implantez en Python cet algorithme sous la forme d'une fonction `algo_dicho (n)`, le paramètre `n` donnant le nombre de boules disponibles.

Exercice 2 *Trier une pile de crêpes*

Nous considérons dans cet exercice un problème simple, dont l'intérêt scientifique ne saute sans doute pas aux yeux. Dans sa version la plus imagée, le problème consiste en effet à mettre dans l'ordre une pile de crêpes de différentes tailles. La crêpe de plus grand diamètre doit se retrouver en dessous de pile ; la plus petite au sommet ; entre les deux, toute crêpe doit reposer sur une crêpe plus large. Pour modifier l'arrangement de la pile de crêpes, vous êtes armé d'une spatule appropriée et contraint à un seul type d'action : insérer la spatule entre deux crêpes et retourner l'ensemble des crêpes du dessus[1].

On représentera les piles de crêpes par des listes des diamètres de ces crêpes (en mm par exemple), le diamètre de la crêpe au sommet de la pile étant le premier élément de la liste.

Il nous faut maintenant une procédure qui simule l'action du retournement d'une partie de la pile de crêpes à l'aide de la spatule. C'est l'objet de la question qui suit.

Question 1 Réalisez une procédure `renverser_sommet` à deux paramètres : une liste ℓ et un entier k compris entre 0 et n, n étant la longueur de la liste, qui renverse l'ordre des k premiers éléments de ℓ. Cette procédure modifie la liste passée en paramètre et ne renvoie rien comme le montre la session qui suit.

```
>>> l = [3,1,4,5,6,2]
>>> renverser_sommet (l,5)
>>> l
[6, 5, 4, 1, 3, 2]
>>> renverser_sommet (l,6)
```

1. D'après un article publié à l'adresse `https://interstices.info/jcms/n_52318/genese-dun-algorithme` dans lequel on apprend que l'algorithme de cet exercice est utilisé pour résoudre des problèmes de routage dans les réseaux de processeurs qui calculent en parallèle.

```
>>> l
[2, 3, 1, 4, 5, 6]
```

Armé de cette procédure vous allez pouvoir trier des piles de crêpes.
L'idée de cet algorithme consiste à
— repérer où se trouve la plus grande des crêpes ;
— insérer la spatule sous cette crêpe et renverser le sommet de la pile ;
— renverser la totalité de la pile de crêpe.
Ayant accompli ces trois points, la plus grande des crêpes se trouve tout en bas de la pile, et il ne reste plus qu'à trier les crêpes situées au dessus.

Question 2 En considérant la pile de crêpes représentée par la liste `l = [5, 1, 4, 2, 6, 3]`, donnez la série des renversements à effectuer en précisant à côté de chacun d'eux l'état de la pile obtenue.

Question 3 Réalisez une procédure de tri d'une pile de crêpes.

```
>>> l = [35, 12, 44, 50, 61, 25]
>>> tri_pile_crepes (l)
>>> l
[12, 25, 35, 44, 50, 61]
```

Exercice 3 *Autour du tri insertion*

Dans cet exercice,
— l'expression *liste triée* signifie liste triée dans l'ordre croissant (et suppose implicitement que les éléments de cette liste sont tous de même type ordonnable) ;
— la complexité d'un algorithme de tri est mesurée en nombre de comparaisons d'éléments de la liste à trier ;
— la méthode `sort` des tris ne peut pas être utilisée.
L'algorithme 1 est un rappel de l'algorithme d'insertion de l'élément d'indice i dans une liste ℓ vu en cours.

Algorithme 1 Algorithme d'insertion

Entrée : ℓ une liste de longueur n telle que $\ell[0:i]$ soit triée, et $0 <= i < n$ un indice
Sortie : la tranche $\ell[0:i+1]$ est triée

```
aux ← ℓ[i]
k ← i
tant que k ≥ 1 et aux < ℓ[k − 1] faire
    l[k] ← l[k − 1]
    k ← k − 1
fin tant que
l[k] ← aux
```

Dans la suite on suppose que cet algorithme a été réalisé en Python dans une fonction `inserer` qui modifie la liste passée en paramètre.

Question 1 En utilisant la procédure `inserer`, réalisez en Python la fonction nommée `tri_insertion` qui modifie la liste passée en paramètre en triant ses éléments.

Question 2 Rappelez la complexité du tri par insertion en fonction de la longueur n de cette liste, en distinguant le meilleur et le pire des cas.

On considère maintenant le problème suivant :

> **Données :** on dispose d'une liste ℓ_1 triée de longueur n, et d'une liste ℓ_2 de longueur m non nécessairement triée. On suppose que $m < n$.
>
> **But :** on souhaite construire une liste triée ℓ_3 de longueur $n+m$ contenant tous les éléments des deux listes données.

Nous allons envisager plusieurs solutions à ce problème.

Solution 1 : elle consiste à construire une liste obtenue par concaténation des deux listes ℓ_1 et ℓ_2 et la trier par insertion en utilisant la fonction `tri_insertion`.

Question 3 Réalisez cette solution en écrivant une fonction nommée `solution1` qui renvoie la liste ℓ_3 demandée, les deux listes ℓ_1 et ℓ_2 étant passées en paramètre.

Question 4 Quelle est, en fonction de n et m, la complexité de cette solution ? Distinguez meilleur et pire des cas.

Solution 2 : elle est une variante de la solution précédente qui tient compte du fait qu'après concaténation des deux listes ℓ_1 et ℓ_2, les n premiers éléments sont déjà dans l'ordre croissant : il n'est donc pas nécessaire de faire tout le travail effectué pas la fonction `tri_insertion`.

Question 5 Réalisez une fonction nommée `solution2`, ayant les mêmes spécifications que `solution1` mais qui n'utilise pas la fonction `tri_insertion` pour exploiter le fait que les n premiers éléments sont déjà dans l'ordre croissant. Cette fonction peut utiliser la procédure `inserer`.

Question 6 Quelle est la complexité de cette solution ? Distinguez meilleur et pire des cas.
