Devoir Surveillé 2 - Corrigé

1 Un problème de rendu de monnaie

On s'intéresse ici au nombre de manières différentes de rendre une somme n en fonction d'une liste de types de pièces disponibles stockés dans une liste L. On considérera que l'on dispose de suffisamment de pièces de chaque type pour rendre la somme n.

On va écrire une fonction `pieces(n,L)` qui va renvoyer le nombre de manières différentes d'obtenir n avec des pièces ayant des valeurs dans L.

Question 1. *Montrer que `pieces(3,[1,2])` vaut 2. Que vaut `pieces(2,[0.5,1])` ?*

On peut écrire $3 = 3 \times 1$ ou $3 = 2 + 1$, ce qui donne deux possibilités. Dans ce problème, il est clair que l'ordre est indifférent. On peut donc noter les pièces par ordre de valeur décroissante.

On peut écrire $2 = 2 \times 1$ ou $2 = 1 + 2 \times 0.5$ ou encore $2 = 4 \times 0.5$. On a donc `pieces(2,[0.5,1])`=3.

Il est par contre clair que si $0 \in$ `L`, le nombre de possibilités devient infini ! En effet un appel à `pieces(2,[2,0])` pourrait renvoyer toute écriture de la forme $2 = 1 \times 2 + m \times 0$ avec $m \in \mathbb{N}$

Question 2. *Compléter les `return` du programme suivant :*

```
def pieces(n,L):
    if n==0: # (1)
        return 1
    elif L == [] or n < 0:
        return 0 # (2)
    else:
        return pieces(n, L[1:]) + pieces(n-L[0], L) # (3)
```

(1) Quand on arrive à $n = 0$, cela correspond à un succès dans le rendu de monnaie : on a épuisé n en puisant dans la liste L.
(2) Quand on arrive à une situation d'impasse, ce qui est le cas si l'on doit rendre la monnaie restante sans plus disposer d'aucune pièce (liste vide) ou que les pièces disponibles sont de valeurs supérieures à la somme à rendre, on renvoie 0, c'est-à-dire que l'on n'ajoute pas 1 à l'addition récursive.
(3) On partitionne les rendus de monnaie en deux parties : les situations où l'on n'utilise pas la valeur `L[0]` et celles où l'on prend une pièce de valeur `L[0]`.

Question 3. *Montrer que ce programme se termine.*

Le programme se termine... si $0 \notin L$. C'est la mise en garde de la première question. On note qu'à chaque appel récursif, la quantité `n + len(L)` est strictement décroissante et que les deux conditions de sorties permettent de ne pas créer d'appel infni : quoi qu'il arrive, on enlève à L des éléments et `len(L)` est une suite décroissante positive. Les valeurs successives de `n` sont aussi décroissantes (au sens large) et il peut arriver que la soustraction `n-L[0]` fasse passer la valeur en-dessous de 0, mais le cas est traité.

2 Transformations récursives de listes

Question 4. *Écrire une fonction récursive `swap2` qui prend en argument une liste `L` et qui renvoie cette liste où l'ordre des éléments est modifié de sorte à ce que cette liste contienne : le second élément de L puis le premier, puis le quatrième, puis le troisième, puis le sixième, puis le cinquième etc...*
Ainsi, `swap2([1,4,9,16,25,36,49])` renverra `[4,1,16,9,36,25,49]`.

```
def swap2(L):
    if len(L) < 2:
        return L
    return [L[1], L[0]] + swap2(L[2:])
```

Question 5. *Réécrire cette fonction dans un cas plus général : écrire une fonction récursive `swap` qui prend en arguments une liste `L` et un entier strictement positif `k` qui renvoie la liste constituée du $k^{ème}$ élément de L, puis du $k-1^{ème}$, ..., puis du premier, puis du $2k^{ème}$ ... du $k+1^{ème}$ etc...*
Ainsi `swap([1,4,9,16,25,36,49],3)` renverra `[9,4,1,36,25,16,49]`

```
def swapp(L, k):
    if len(L) < k:
        return L
    M = L[:k]
    M.reverse()
    return M + swapp(L[k:], k)
```

```
def swap(L, k):
    if len(L) < k:
        return L
    return L[k-1::-1] + swap(L[k:], k)
```

L'énoncé est un peu ambigu en ce qu'il ne précise pas s'il faut inverser la queue de la liste (les `len(L)%k` éléments qui restent à la fin après avoir épuisé les découpages possibles en sous-listes de longueur k. On a choisi ici de la laisser à l'identique. S'il faut l'inverser aussi, il suffit de remplacer le `return L` de la condition de sortie par `return L[::-1]`.

La première solution utilise la méthode `reverse`, qui retourne une liste en place. Elle nécessite donc de commencer par extraire la sous-liste à inverser, puis de l'inverser, avant de pouvoir enfin écrire un `return`. En effet, comme c'est une méthode, cela n'aurait aucun sens d'écrire `return L.reverse()`.

La deuxième solution utilise toute la puissance du slicing et de ses trois arguments. On aurait pu aussi remplacer `L[k-1::-1]` par `list(reversed(L[:k]))`, mais cette solution, quoique rapide à l'écriture, est lente à l'exécution car elle commencer par créer un itérateur à partir duquel elle crée une liste.

Question 6. *On considère dans cette question que l'affectation ou l'ajout d'un élément à une liste a un coût de 1, et qu'un slicing de liste a un coût égal à la longueur de la liste slicée : ainsi `L[a:b]` a un coût de $b-a$ et `L[:]` a un coût de $n =$ `len(L)`.*

Exprimer et justifier la complexité (asymptotique) de l'algorithme écrit précédemment en fonction de n et de k. Si besoin, on pourra considérer que n est grand devant k.

Prenons la variante avec le slicing. Chaque passage dans la fonction récursive a un coût de $k+2$, avec k pour le slicing, 1 pour la comparaison et 1 pour l'addition. Il y a $\lfloor n/k \rfloor$ passages dans la boucle amenant la transformation d'une liste de longueur k, le dernier passage (qui a lieu si n n'est pas multiple de k) a un coût de 1 (pas d'inverse) ou de $n\%k+1$ dans le cas contraire. Bref, on peut dire que le coût est en $\mathcal{O}(k \times (n/k)) = \mathcal{O}(n)$ et ne dépend donc pas de k.

3 Question de cours : tri par insertion

Question 7. *Écrire une fonction `tri_insertion(L)` qui prend en argument une liste de nombres `L` et la trie dans l'ordre croissant en utilisant l'algorithme du tri par insertion.*

```
def tri_insertion(L):
    for i in range(1, len(n)):
        j = i
        while L[j-1] > L[j] and j > 0:
            L[j-1], L[j] = L[j], L[j-1]
            j -= 1
    return L
```

```
def tri_insertion_rec(L, L1):
    if L1 == []:
        return L
    L.append(L1.pop(0))
    j = len(L) - 1
    while L[j-1] > L[j] and j > 0:
        L[j-1], L[j] = L[j], L[j-1]
        j -= 1
    return tri_insertion_rec(L,L1)
```

La version de gauche est la version standard. Celle de droite, qui n'était pas demandée (nous y avons pensé trop tard) est récursive. Pour trier la liste `L`, il faut rentrer `tri_insertion_rec([], L)`. Au fil de l'exécution, `L` stocke la partie triée et `L1` celle restant à trier.

Question 8. *Donner sans justification la complexité dans le pire et le meilleur des cas du tri par insertion en fonction de* `n` *la longueur de la liste* `L` *à trier.*

La complexité est en $\mathcal{O}(n^2)$ dans le pire des cas (liste triée par ordre décroissant) et linéaire dans le meilleur des cas (liste déjà triée par ordre croissant).

4 Tri à bulles

Le tri à bulles consiste à effectuer dans un premier temps $n-1$ comparaisons et les échanges correspondants nécessaires entre éléments consécutifs permettant de placer le plus grand élément d'une liste de longueur n à trier en queue de celle-ci. Si un échange a été effectué, un booléen est mis à vrai ; sinon, il lui est affecté la valeur « faux ».

Tant que ce booléen est vrai, on recommence cet algorithme en effectuant $n-2$ comparaisons et échanges éventuels et en plaçant le plus grand élément restant en avant-dernière position et ainsi de suite avec $n-3$ comparaisons et échanges éventuels etc...

Question 9. *Écrire une fonction* `tri_bulle(L)` *qui trie dans l'ordre croissant une liste* `L` *comme décrit précédemment.*

```
def tri_bulle(L):
    n = len(L)
    perm = True
    while perm and n > 0:
        perm = False
        for j in range(n-1):
            if L[j] > L[j+1]:
                L[j+1], L[j] = L[j], L[j+1]
                perm = True
        n -= 1
    return L
```

Question 10. *Montrer que cet algorithme s'arrête.*

À chaque parcours de la boucle `while`, la variable n est diminuée d'une unité. Comme la condition du `while` porte (notamment) sur le fait que $n > 0$, on effectue au plus n passages dans cette boucle conditionnelle. n étant entier, strictement décroissant et garantissant l'arrêt de l'algorithme lorsqu'il devient négatif, c'est un variant de boucle.

Question 11. *Quelle est la complexité de cet algorithme en fonction de n, où n est la taille de* `t`*, dans le pire des cas ? Justifier la réponse.*

Dans le pire des cas, la condition sur `perm` ne permet pas de sortir prématurément de la boucle conditionnelle. Le k-ème passage dans cette boucle entraîne un passage dans la boucle `for` avec la valeur $n-k-1$ et un coût équivalent (en négligeant les affectations et en ne prenant en compte que les comparaisons). On obtient ainsi une complexité en $\mathcal{O}(n+(n-1)+(n-2)+\cdots+1) = \mathcal{O}(n^2)$, comme dans le tri par insertion.

Question 12. *Quelle est la complexité de cet algorithme dans le meilleur des cas ? Quelles sont les listes correspondantes ?*

Dans le meilleur des cas, le booléen `perm` fait sortir de la boucle `while` après le premier passage. Cela veut dire que l'on n'a procédé à aucun échange (corps du test `if`), donc que la liste était déjà triée. La complexité est alors linéaire ($n-1$ comparaisons exactement.

Question 13. *Montrer que cet algorithme trie effectivement* `L` *par ordre croissant. On utilisera un invariant de boucle et l'on remarquera qu'après k passages dans la boucle principale du programme, les k derniers éléments sont les plus grands de la liste* `L` *et sont triés par ordre croissant.*

Si l'on ignore le booléen, le tri proposé consiste, en parcourant la liste par indices croissants, à placer par échanges successifs le plus grand élément en bout de liste, puis à recommencer sur `L[:n-1]`, etc. Au bout de k passages, les k plus grands éléments sont en bout de liste et classés.

Le booléen permet d'interrompre le processus avant $k = n$ en remarquant que, si le fait de placer le k-ème plus grand élément en bout de `L[:n-k+1]` n'a occasionné aucun échange, c'est que toute la liste est en fait classée et que l'on peut donc s'arrêter.

Ceci suggère l'invariant de boucle suivant :

(P_k) : Après k passages dans la boucle `while`, les k derniers éléments sont les k plus grands éléments de L et sont triés dans l'ordre croissant ; Si `perm == False`, la liste est complètement triée.

- L'initialisation est immédiate.

- Hérédité : Supposons (P_k) vraie.
  Le booléen `perm` est mis à faux. Si `L[j] <= L[j+1]` pour tout j inférieur à $n-2$, alors la liste `L[:n]` est triée. Or, $n =$ `len(L)-k-1` et les k derniers éléments de `L` sont triés et tous supérieurs aux éléments de `L[:n]` ; cela signifie que les `len(L)-n-1` derniers éléments de L sont triés et supérieurs aux précédents. Comme `L[n:]` contient `len(L)-1-n` éléments, cela revient à dire que `L[n:]` est à la fois triée et composée d'éléments supérieurs à ceux de `L[:n]`, donc que `L` est triée ; dans ce cas, (P_{k+1}) est vraie.
  Sinon, `perm == True`. Il y a donc eu au moins un échange d'éléments et, à chaque comparaison d'éléments successifs, l'élément le plus grand est remonté ; le plus grand des éléments de `L[:n]` est donc placé en position $n-1$. Comme il est inférieur aux éléments suivants, les $k+1$ derniers éléments de `L` sont les plus grands de `L` et sont triés ; et dans ce cas, (P_{k+1}) est vraie également.
  L'hérédité est ainsi établie.

- Correction partielle : cet algorithme a deux façons de s'arrêter. Soit `perm == False` ; soit `n == 0`.
  Si `perm` est faux, par hypothèse de récurrence, la liste est complètement triée et la correction partielle établie.
  Si `n == 0`, comme n décroît d'une unité à chaque passage de boucle et qu'initialement, on avait $n =$ `len(L)`, il y a eu `len(L)` passages dans la boucle. Par hypothèse de récurrence, les `len(L)` derniers éléments de `L` sont alors triés par ordre croissant, ce qui signifie que `L` est triée et la correction partielle établie une fois de plus.

La propriété annoncée est donc bien un invariant de boucle ; la correction partielle de l'algorithme est démontrée. Comme on a démontré précédemment que cet algorithme s'arrête, on vient de prouver la correction totale de l'algorithme, qui trie effectivement la liste dans l'ordre croissant.

On notera que la rédaction formelle par invariant de boucle peut être un peu délicate à rédiger. Un étudiant de niveau modeste pourrait à la rigueur se contenter de la première partie et espérer être noté correctement. Un étudiant plus ambitieux, et a fortiori un étudiant désirant se spécialiser en informatique en école aura intérêt à essayer la rédaction plus formelle par invariant de boucle.