1ère NSI — Exercices d'Entraînement

Algorithmique

Lycée Fustel de Coulanges, Massy

Marc Biver, février 2024, *v0.2*

Version incluant les réponses aux exercices.

Exercice 1: Docstring & Tests

Soit la fonction de calcul de puissance suivante (dont on a étudié l'algorithme en cours) :

```
def Puissance(x, n):
    r = 1
    for i in range(n):
        r = r * x
    return r
```

a. Rédiger la docstring d'une telle fonction ;

b. Déterminer un jeu de tests pour cette fonction – en d'autres termes, compléter le tableau suivant qui liste les couples (x, n) qu'il conviendra d'utiliser pour convenablement tester cette fonction.

x	n	Résultat attendu
2	2	4
.....	.....	.....

```
def Puissance(x, n):
    '''
    Fonction qui prend en entrée deux entiers naturels et renvoie le premier
    élevé à la puissance du second.
    '''
    r = 1
    for i in range(n):
        r = r * x
    return r
```

Il n'y a pour le jeu de tests pas "une" bonne réponse, mais ce qu'on va chercher à faire c'est effectuer au moins deux tests de valeurs "normales" dont un avec une grande valeur (en plus, donc, du (2,2) de l'énoncé), ainsi que des tests avec les valeurs aux limites – en l'occurrence 0 et 1 – associées à des valeurs choisies au hasard. On pourra donc proposer quelque chose comme :

x	***n***	***Résultat attendu***
2	*2*	*4*
27	*7*	*10.460.353.203*
21	*0*	*1 (tout nombre élevé à la puissance 0 vaut 1)*
35	*1*	*35*
1	*12*	*1*
0	*99*	*0*
0	*0*	*1 (tout nombre élevé à la puissance 0 vaut 1 – même 0)*

Exercice 2: Variant & invariant de boucle

Soit la fonction suivante :

```
1  def SommeEltLst(liste):
2      '''
3      Fonction qui prend en entrée une liste de nombres et qui renvoie la
4      somme des nombres qui la constituent.
5      '''
6      res = 0
7      for i in range(len(liste)):
8          res += liste[i]
9
10     return res
```

a. Donner un variant de boucle pour cette fonction et montrer qu'elle se termine systématiquement ;

> **Rappel:**
>
> On rappelle qu'un variant de boucle est une quantité entière positive qui décroît strictement à mesure qu'on passe dans la boucle, ce qui permet de justifier que la boucle, tôt ou tard, se terminera.

b. Donner un invariant de boucle pour cette fonction et montrer qu'elle est correcte.

> **Rappel:**
>
> On rappelle qu'un invariant de boucle est une quantité ou une propriété qui est vraie avant et après chaque itération de la boucle – et en particulier *avant* que l'on rentre dans la boucle, et *après* sa dernière itération. Il permet de justifier que le résultat voulu sera atteint. On procède pour cette technique en quatre étapes :
>
> (a) On choisit l'invariant :
>
> — Comprendre clairement le but de la boucle – qu'est-elle censée accomplir ? Quel est le résultat attendu ?
>
> — Partir "de la fin", c'est-à-dire du résultat attendu et identifier quelle quantité est "construite" au fur et à mesure des itérations de la boucle pour constituer ce résultat.
>
> — Ceci devrait vous mettre sur la voie de votre invariant – une propriété (somme d'éléments déjà traités, ordre d'éléments dans une liste...) qui ne change pas malgré les itérations de la boucle.
>
> (b) On montre que l'invariant est vérifié avant la boucle (initialisation) ;
>
> (c) On montre que si l'invariant est vérifié *avant* un passage dans la boucle, alors il est préservé *après* le passage dans la boucle ;
>
> (d) On peut conclure sur la valeur finale à la sortie de la boucle.

a. *Variant de boucle – on est dans le cas ultra-classique du parcours d'une liste par indice, le variant est donc la quantité $len(liste) - i - 1$: il est en effet positif au départ (au pire nul si la liste est vide), et il décroît strictement à chaque itération de la boucle puisque $len(liste)$ reste constant et que i est incrémenté de 1 à chaque fois. Il va donc atteindre 0, ce qui terminera la boucle.*

b. *Décomposons la démarche :*

   (a) *Choix de l'invariant : le but de la boucle est le calcul de la somme des éléments de la liste, et c'est la variable* `res` *qui est construite pour atteindre ce résultat : à chaque étape elle contient la somme des éléments d'indice de 0 à* `i` *de la liste. On peut donc dire que l'invariant de boucle est : "la variable* `res` *contient la somme des éléments d'indice de 0 à* `i` *de la liste" ;*

   (b) *Avant la première itération de la boucle,* `i` *vaut 0 et* `res` *aussi – donc on peut dire que l'invariant est vérifié ;*

   (c) *Si res contient les i premiers éléments de la liste et que l'on passe dans la boucle une fois supplémentaire,* `i` *et* `res` *deviennent respectivement : $i' = i + 1$ et $res' = res + liste[i'] = res + liste[i + 1]$ – et donc l'invariant est bien vérifié à la fin de l'itération de la boucle.*

   (d) *A la fin de la boucle i vaut $(n - 1)$ donc l'invariant s'exprime "la variable* `res` *contient la somme des éléments d'indice de 0 à* `(n-1)` *de la liste" – soit tous les éléments de la liste, et donc la fonction est correcte.*

Exercice 3: Variant & invariant de boucle — suite...

Soit la fonction suivante :

```
1  def RechercheDiviseur(nb):
2      '''
3      Fonction qui prend en entrée un entier strictement supérieur à 1 nb
4      et renvoie son plus petit diviseur autre que 1.
5      Si elle n'en trouve pas (que le nombre est donc premier), elle
6      renvoie 0.
7      '''
8      div = 0
9      i = 2
10     while i < nb:
11         # On vérifie si nb est un multiple de i
12         if nb % i == 0:
13             # Dès qu'on en a trouvé un on le renvoie
14             div = i
15             return div
16         i += 1
17     # On atteindra ce point si on n'a pas trouvé de diviseur et on renverra
       ↪  donc la valeur initiale de div, soit 0.
18     return div
```

a. Donner un variant de boucle pour cette fonction et montrer qu'elle se termine systématiquement ;

b. Donner un invariant de boucle pour cette fonction et montrer qu'elle est correcte.

> **Indice:**
>
> On rappelle qu'un invariant de boucle n'est pas nécessairement une formule mathématique – il peut aussi être une propriété que l'on pourrait exprimer par une phrase du genre (où bien entendu il faut remplir les trous) : "Quel que soit l'entier k tel que 2 ≤ k ≤ ___, k n'est pas___".

c. (question bonus 1) Cette boucle "`while`" semble assez artificielle – on dirait presque qu'elle n'est là que pour vous forcer à travailler sur la notion de variant de boucle... Quelle est la boucle "`for`" équivalente qu'on utiliserait plus logiquement dans un tel cas ?

d. (question bonus 2) Cette fonction n'est pas franchement optimale... Comment remplacer la condition "`while i < nb:`" pour lui faire faire nettement moins de tests tout en gardant le bon résultat ?

a. *Variant de boucle – situation assez simple : la quantité $nb - i$ débute à la valeur $nb - 2$ qui est nécessairement positive puisqu'il est spécifié que nb est strictement supérieur à 1 ; cette quantité décroît strictement à chaque itération de la boucle puisque nb reste constant et que i est incrémenté de 1 à chaque fois. Il va donc atteindre 0, ce qui terminera la boucle.*

b. *Décomposons la démarche :*

   (a) *Choix de l'invariant : le but de la boucle est de trouver le diviseur le plus petit diviseur possible de nb, donc à chaque étape de la boucle, aucun nombre plus petit que celui que l'on est en train de considérer n'est diviseur de nb. On peut donc exprimer l'invariant : "quel que soit l'entier k tel que 2 ≤ k ≤ (i-1), k n'est pas un diviseur de nb" ;*

   (b) *Avant la première itération de la boucle, `i` vaut 2, donc il n'existe aucun entier k tel que 2 ≤ k ≤ (i-1) – donc on peut dire que l'invariant est vérifié ;*

   (c) *Si l'invariant est vérifié pour une valeur de i, alors pour la valeur suivante $i' = i + 1$, deux cas sont possibles :*

   - *Soit i est un diviseur de nb et la fonction le renvoie et donc se termine (donc la question de l'invariant ne se pose plus) ;*
   - *Soit i n'est pas un diviseur de nb – et la boucle se poursuit et l'invariant est vérifié : "quel que soit l'entier k tel que 2 ≤ k ≤ (i'-1), k n'est pas un diviseur de nb" (puisque $i' - 1 = i$).*

   (d) *A la fin de la boucle, dans l'hypothèse où on n'est pas sorti de la fonction, i vaut nb, donc l'invariant s'exprime "quel que soit l'entier k tel que 2 ≤ k ≤ (nb-1), k n'est pas un diviseur de nb" – et donc nb est par définition un nombre premier.*

c. *On fait varier i de 2 à nb-1 donc la boucle correspondante s'écrirait "`for i in range(2, nb)`".*

d. *Il est évident que le plus grand diviseur d'un nombre (autre que lui-même) est sa moitié s'il est pair, ou son tiers s'il est impair mais un multiple de trois, etc. En tout état de cause, aucun nombre ne pourra avoir de diviseur strictement supérieur à sa moitié (ou la partie entière de sa moitié dans le cas d'un nombre impair). En écrivant la condition "`while i <= (nb`*

// 2)" on économiserait presque la moitié des opérations effectuées par la boucle tout en maintenant le résultat correct...

Exercice 4: Variant & invariant de boucle — encore un !

Soit la fonction suivante :

```
1  def RechercheMax(Lst):
2      '''
3      Fonction qui prend en entrée une liste de nombres positifs
4      et renvoie la valeur du plus grand d'entre eux.
5      '''
6      res = 0
7      for i in range(len(lst)):
8          if lst[i] > res:
9              res = lst[i]
10
11     return res
```

Donner un invariant de boucle pour cette fonction et montrer qu'elle est correcte.

1. *Choix de l'invariant : le but de la boucle est de trouver le maximum de la liste, donc avant chaque étape de la boucle, la variable* `res` *contient le maximum des éléments d'indice de 0 à* `i-1` *de la liste ;*
2. *Avant la première itération de la boucle,* `i` *vaut 0, donc la liste d'éléments dont l'indice est compris entre 0 et (i-1) est vide – donc on peut dire que l'invariant est vérifié ;*
3. *Si l'invariant est vérifié pour une valeur de i, alors pour la valeur suivante* $i' = i + 1$*, deux cas sont possibles :*
   - *Soit lst[i]* $\leq$ *res et donc, par définition, l'invariant reste vérifié (le maximum n'a pas changé) ;*
   - *Soit lst[i]* $>$ *res, res prend alors la valeur lst[i] qui est le nouveau maximum des éléments d'indice compris entre 0 et (i'-1) (donc i), et donc l'invariant est vérifié aussi.*
4. *A la fin de la dernière itération de la boucle, i vaut len(lst), donc res contient le maximum des éléments d'indice de 0 à* `len(lst)-1` *de la liste – donc de toute la liste, et donc la boucle est correcte.*

Exercice 5: Complexité – inversion de liste

Soit la fonction suivante :

```python
def InverseListe(lst):
    '''
    Fonction qui prend en entrée une liste quelconque et qui renvoie la
    ↪  liste inversée.
    '''
    # On initialise la liste résultat - liste vide
    res = []
    # On la remplit en partant de la fin de la liste en entrée
    for i in range(len(lst)):
        res.append(lst[len(lst) - i - 1])

    return res
```

a. Montrer, en utilisant un invariant de boucle, qu'elle fait bien ce que sa docstring spécifie.

b. Compter son nombre d'opérations élémentaires, et en déduire sa complexité (préciser sa dénomination et sa notation en "$\mathcal{O}$").

a. *L'invariant de boucle est la propriété, pour une valeur de i et après la réalisation de la boucle : "res est une liste contenant les i+1 derniers éléments de lst, dans l'ordre inverse". Initialisation : i=0, res contient uniquement lst[len(lst) - 1], soit le dernier élément de lst, donc l'invariant est vérifié. Si c'est vérifié pour i, lorsqu'on passe à i' = i + 1, on ajoute à la fin de res l'élément de lst d'indice (len(lst) - i' - 1) qui est le (i' + 1)^{ème} en partant de la fin, et on a donc rallongé res de 1 (sa longueur est donc maintenant i'+1) – l'invariant est donc bien vérifié à i'. La boucle se termine lorsque i = après len(lst) itérations, quand i est égal à len(lst) - 1, et donc à ce moment l'invariant s'exprime "res est une liste contenant les (len(lst)) derniers éléments de lst, dans l'ordre inverse" – donc tous les éléments, et donc la boucle est correcte.*

b. *Si l'on compte strictement les opérations on a :*
   - *Deux opérations en dehors de la boucle (`res = []` et `return res`) ;*
   - *Une opération à l'intérieur de chaque itération de la boucle (le append sur la lsite res) ;*
   - *Deux opérations à chaque itération de la boucle – l'incrémentation de i et sa comparaison avec len(lst).*

   Si on note n la longueur de lst, la boucle est réalisée n fois et donc le nombre d'opérations est : $3 \times n + 2$ – et donc on en déduit que la complexité est en $\mathcal{O}(n)$, c'est-à-dire linéaire.

Exercice 6: Complexité – note pondérée

Soit la fonction suivante :

```python
def EltPondere(lst_notes, indice):
    '''
    Fonction qui prend en entrée une liste de notes et un indice dans cette
    ↪  liste et qui renvoie la valeur pondérée de cette note (donc sa
    ↪  valeur divisée par le nombre de notes).
    '''
    res = lst_notes[indice] / len(lst_notes)
    return res
```

Déterminer sa complexité.

Aucun piège ici – certes, on utilise len(lst_notes) dans la fonction, mais pas dans le cadre d'une boucle. On n'exécute donc ici que deux opérations : le calcul de res et le return. On est donc dans une complexité constante, soit $\mathcal{O}(1)$.

Exercice 7: Complexité – somme de produits ou produit de sommes ?

Soit les deux fonctions suivantes :

```python
def ProdSomme(n1, n2):
    '''
    Fonction qui prend en entrée deux entiers n1 et n2 et renvoie
    le produit des sommes de tous les i et j pour i < n1 et j < n2.
    '''
    res1 = 0
    res2 = 0
    for i in range(n1):
        res1 += i
    for j in range(n2):
        res2 += j

    res = res1 * res2

    return res
```

```python
def SommeProd(n1, n2):
    '''
    Fonction qui prend en entrée deux entiers n1 et n2 et renvoie
    la somme de tous les produits i.j pour i < n1 et j < n2.
    '''
    res = 0
    for i in range(n1):
        for j in range(n2):
            res += i * j
            #On rappelle que "a += b" est équivalent à "a = a + b"

    return res
```

a. Démontrer (simplement) que si j'applique ces deux fonctions aux mêmes paramètres n1 et n2 leur retour sera identique – qu'en d'autres termes elles font le même calcul.

b. Quelles sont les nombres d'opérations élémentaires qu'effectuent chacune de ces deux fonctions (exprimés en fonction de n1 et n2) ? Laquelle des deux, du coup, vous semble la meilleure pour atteindre le résultat ?

a. *On voit de manière évidente que si l'on développe le produit de ProdSomme on tombera sur la somme de SommeProd :*

ProdSomme renvoie : $[1+2+(...)+(n_1-1)] \times [1+2+(...)+(n_2-1)]$
SommeProd renvoie : $1 \times 1 + 1 \times 2 + (...) + (n_1-1) \times (n_2-1)$

b.
- *ProdSomme effectue successivement deux boucles de longueurs respectives n1 et n2 – donc le nombre d'opérations élémentaires qu'elle va effectuer est de l'ordre de* $n1 + n2$ [a].
- *SommeProd, lui, effectue deux boucles imbriquées : la boucle intérieure (`for j in range(n2)`) va effectuer n2 itérations tandis que la boucle extérieure (`for i in range(n1)`) va en effectuer n1 – donc le nombre d'opérations élémentaires qu'elle va effectuer est de l'ordre de* $n1 \times n2$ [a].

Il est donc évident que, pour des valeurs importantes de n1 et n2, ProdSomme sera beaucoup moins complexe (et donc beaucoup plus performante) que SommeProd – et ce pour le même résultat. Si on choisit par exemple n1 = 1.000.000 (un million) et n2 = 2.000.000 alors ProdSomme effectuera environ $n1 + n2 = 3.000.000$ *d'opérations, tandis que SommeProd en effectuera environ* $n1 \times n2 = 2.000.000.000.000$ *(deux mille milliards) – soit plus de 650.000 fois plus !*

a. Notez qu'ici on parle bien d'ordre de grandeur et qu'on néglige les opérations unitaires (comme le `res = 0` ou `return res`) qui ne changeront rien à la performance de la fonction quand on commencera à utiliser de grandes valeurs de n1 et n2.

Exercice 8: Tri à bulles - bidouillons-le un peu...

On rappelle le code Python du tri à bulles dans sa version la plus basique :

```
1  def TriBulles(liste):
2      '''
3      Fonction qui effectue le tri par permutations de la liste passée en
       ↪ entrée.
4      '''
5      n = len(liste)
6      for i in range(n):
7          for j in range(n-1):
8              if liste[j] > liste[j+1]:
9                  # Cette syntaxe permet d'affecter deux variables
                   ↪ simultanément et donc de ne pas passer par une variable
                   ↪ intermédiaire
10                 liste[j], liste[j+1] = liste[j+1], liste[j]
11     return liste
```

a. A quoi sert l'indice "i" dans cette fonction ?

b. Que se passe-t-il si on remplace la ligne 7 par `for j in range(n-1, -1, -1)` ? Cette syntaxe – qui n'est rien de plus qu'une utilisation de la syntaxe `range` habituelle mais dans le contexte d'une suite décroissante) permet de réaliser une boucle allant de n-2 (le premier paramètre) à 0 (arrêt juste avant d'atteindre le deuxième paramètre) par pas de -1 (le troisième paramètre). L'indice va donc décrire la suite (n-2), (n-3), ..., 1, 0 au lieu de 0,

1, ..., (n-3), (n-2).

c. Que se passe-t-il si on remplace la ligne 8 par `if liste[j] < liste[j+1]:` ?

d. La boucle interne ne fait-elle pas des opérations inutiles ? Comment la modifier pour la rendre plus efficace ?

a. Uniquement à compter le nombre de fois que l'on fait tourner la boucle interne ! La variable i, comme on peut le constater, n'est utilisée nulle part hormis dans le `for` de la boucle.

b. Eh bien... Rien du tout sur le résultat : on continue à échanger les éléments si celui de droite (d'indice supérieur) est inférieur à celui de gauche, donc on met à terme la liste dans le même ordre que précédemment. En revanche, on s'y prend dans l'autre sens, ce qui modifie la démarche : au lieu de faire "remonter" le plus grand élément du début à la fin de la liste, on va faire ici "descendre" le plus petit de la fin au début. Concrètement, si on passe en entrée la liste qu'on a utilisée en cours (`[2, 5, 3, 1]`) on passera par les étapes suivantes :

(a) `[2, 5, 3, 1]`
(b) `[1, 2, 5, 3]`
(c) `[1, 2, 3, 5]`

*c. Là, en revanche, on inverse le sens de tri – on fait une permutation si l'élément de droite (d'indice supérieur) est **supérieur** à celui de gauche : donc on obtiendra à la fin la liste triée par ordre décroissant au lieu de croissant.*

d. A la fin de la première itération de la boucle externe (donc quand la boucle interne a fini son travail une première fois), on a ramené le plus grand élément de la liste à la fin de celle-ci ; à la fin de la deuxième itération, le deuxième plus grand est en avant-dernière position ; (...) ; à la fin de la $i^{ième}$ itération, le $i^{ième}$ plus grand est en $i^{ième}$ position en partant de la fin, et tous ceux qui le suivent sont positionnés en ordre croissant[a]*. Donc quand on passe à la $(i+1)^{ième}$ itération de la boucle, on a déjà les i éléments "de droite" (de plus grands indices) de la liste qui sont déjà triés et qu'il ne sert plus à rien, de trier de nouveau – on peut donc modifier la boucle interne pour qu'elle ne regarde plus ces éléments, et donc soit plus optimale, en la faisant aller non plus de 0 à n-2 mais de 0 à n-i-2. Le résultat sera le suivant :*

```python
def TriBullesOptim(liste):
    '''
    Fonction qui effectue le tri par permutations de la liste passée en entrée.
    '''
    n = len(liste)
    for i in range(n):
        # On ne regarde dans la boucle interne que les éléments qui ne sont pas déjà triés
        for j in range(n-i-1):
            if liste[j] > liste[j+1]:
                liste[j], liste[j+1] = liste[j+1], liste[j]
    return liste
```

---

a. Si vous ne me croyez pas, démontrez-le en utilisant un invariant de boucle – c'est exactement à cela qu'ils servent !!