Corrigé Sujet 1 épreuve pratique NSI

Michel Billaud (michel.billaud@laposte.net)

30 janvier 2022

Table des matières

- mise à jour 1er février, complément sur les versions récursives
- 2 février, ajout versions itératives du rendu de monnaie.

1 Licence

Cette collection de notes est mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 2.0 France.

- Les notes sont publiées dans https://www.mbillaud.fr/notes/
- Sources dans https://github.com/MichelBillaud/notes-diverses

2 Le sujet

Se trouve sur la page https://eduscol.education.fr/2661/banque-des-epreuves-pratiques-de-specialite-nsi, dans https://eduscol.education.fr/document/33178/download

3 Exercice 1 : nombre d'occurrences d'un caractère dans un mot

3.1 La question

> **Question** : Écrire une fonction `recherche` qui prend en paramètres `caractere`, un caractère, et `mot`, une chaîne de caractères, et qui renvoie le nombre d'occurrences de `caractere` dans `mot`, c'est-à-dire le nombre de fois où `caractere` apparaît dans `mot`.

Clairement, le souhait est de normaliser le choix des noms - ça facilite les corrections -, on ne laisse pas d'autre possibilité que de commencer par

```
def recherche(caractere, mot):
    ...
```

Critique : le nom imposé `recherche` est assez malheureux. L'usage est que le nom d'une fonction décrive le résultat qu'elle calcule. "Recherche", ça ne dit pas ce qui est recherché. La première position du caractère dans le mot ? juste un booléen qui dit si on l'a trouvé ? …

Bref, un nom comme `nombre_occurrences` ou `nb_occurrences` aurait été plus approprié.

3.2 Solution 1 : itération et comptage

Une solution simple, dans le style impératif, est de

- regarder le mot caractère par caractère, avec une boucle (`for c in mot: ...`)
- comparer avec le caractère cherché -`if c == caractere:`
- et si ça colle, ajouter 1 à un compteur (variable locale de la fonction)
  - initialisé à 0 avant la boucle
  - et dont la valeur sera retournée comme résultat de la fonction

```
def recherche(caractere, mot):
    nb = 0                      # nombre d'occurrences, 0 au départ
    for c in mot:               # pour chaque caractère c du mot
        if c == caractere:      #   - est-ce le bon ?
            nb += 1             #   - oui ! un de plus !
    return nb                   # et voilà le total.
```

Je suppose que c'est la solution à laquelle les correcteurs s'attendent le plus. Mais on peut faire autrement.

3.3 Solution 2 : sous-liste en intension et comptage

Une chaîne de caractères est aussi une liste de caractères. À ce titre,

- on peut en extraire facilement la sous-liste de caractères qui nous plaisent (qui sont égaux à ceux qu'on cherche), par `[ c in mot if c == caractere ]`
- et il ne reste plus qu'à retourner la longueur(`len`) de cette liste.

```
def recherche(caractere, mot):
    return len([c for c in mot if c == caractere])
```

3.4 Solution 3 : la méthode `count` des chaînes

Quand on connaît le langage Python, on sait que les chaînes sont des séquences qui ont une méthode `count` qui retourne le nombre d'occurrences d'une sous-séquence.

```
def recherche3(caractere, mot):
    return mot.count(caractere)
```

bref, c'est tout fait.

3.5 Solution 4 : récursion structurelle naïve

Sachant que

- une chaîne est vide ou pas,
- si elle est vide, elle contient 0 occurrence de quoi que ce soit,
- si elle n'est pas vide, elle est composée d'un premier caractère et d'une suite ;
- ce caractère est celui qu'on cherche, ou pas ;

on peut s'orienter vers une solution récursive

```
def recherche(caractere, mot):
    if len(mot) == 0:
        return 0
    else:
        premier = mot[0]
        suite = mot[1:]
        if premier == caractere:
            return 1 + recherche(caractere, suite)
        else:
            return recherche(caractere, suite)
```

qu'on peut présenter de diverses façons équivalentes, plus ou moins détaillées, par exemple en utilisant seulement une variable pour le nombre d'occurrences qui figurent dans la suite :

```
def recherche(caractere, mot):
    if len(mot) == 0:
        return 0
    else:
        nb_occ_suite = recherche(mot[1:])
```

```
        if mot[0] == caractere:
            return 1 + nb_occ_suite
        else:
            return nb_occ_suite
```

ou encore, en jonglant avec les expressions conditionnelles de Python

```
def recherche (caractere, mot):
    return 0 if len(mot) == 0 else (
           recherche(caractere, mot[1:])
              + (1 if caractere == mot[0] else 0))
```

qui se lit :

- retourner 0 si le mot est vide
- ou alors le nombre d'occurrences du caractère dans la suite, avec 1 de plus si le premier caractère du mot était le bon.

3.6 Critique de la solution récursive naïve

Cette solution fournit un résultat correcte, mais elle a un défaut, sa **complexité temporelle**, liée au fait que la construction de `mot[1:]` pour un mot de taille n implique une copie des $n-1$ éléments concernés dans une nouvelle liste.

En gros, si on part d'un mot de 10 lettres, le passage au sous-mot de 9 lettres nécessitera de copier 9 éléments. Mais pour traiter ce mot de 9 lettres, il faudra aussi faire une copie du sous-mot de 8 lettres, et le traiter.

Bref, tout ça va nous coûter 9+8+….+1 étapes. Pour un mot de taille n, ça nous fera $\frac{n(n+1)}{2}$ étapes, une quantité qui grandit comme le carré de n. On parle de complexité temporelle **quadratique**.

Comme nous disposons d'une autre solution (itérative) qui n'est pas compliquée et qui fait le calcul en un temps qui est simplement proportionnel à n (complexité **linéaire**), en pratique, on n'utilisera évidemment pas la solution récursive présentée ci-dessus.

3.7 Solution 5 : Sauvons la récursion !

Ceci ne veut pas dire **du tout** qu'une approche récursive soit **intrinsèquement** inefficace.

Ce qui coûte ici, c'est de faire la **copie** de la fin du mot, pour la passer en paramètre.

À la place, on peut passer le mot inchangé, et un paramètre qui indique l'indice du premier caractère restant à traiter.

Ce qu'on peut présenter sous cette forme

```
def recherche(caractere, mot, debut = 0):
    if debut == len(mot):
        return 0
    else:
        r = recherche(caractere, mot, debut + 1)
```

```
        if caractere == mot[debut]:
            return r + 1
        else:
            return r
```

Ce coup-ci, le temps de calcul est linéaire.

On retrouve cette manière de faire une récursion sur une sous-liste dans l'exercice précédent.

Note : les *expressions conditionnelles* de Python permettent de réduire les 4 dernières lignes à

```
    return  r + 1  if caractere == mot[debut]  else r
```

qui se lit naturellement "retourne r+1 si ceci-cela, et 0 sinon".

3.8 Solution 6 : Récursion avec variable tampon

Profitons de l'occasion pour utiliser la technique de la "variable tampon", qui est également présente dans l'exercice suivant.

En gros, on se sert d'un paramètre supplémentaire qui sert à construire le résultat final. Ici, `total` indiquera le nombre d'occurrences trouvé jusque là.

Quand la récursion est terminée, la fonction retourne le contenu du tampon

```
def recherche(caractere, mot, debut = 0, total = 0):
    if debut == len(mot):
        return total
    t = 1  if caractere == mot[debut]  else 0
    return recherche(caractere, mot, debut + 1, total + t)
```

La technique du paramètre tampon est souvent utilisée pour obtenir un algorithme récursif où la récursion est terminale (c'est-à-dire que l'appel récursif est la dernière chose qu'on fait dans la fonction).

L'intérêt de la récursion terminale, c'est que l'interprète n'a rien à mémoriser dans la pile lors de l'appel, et peut même transformer l'appel récursif en boucle. Comme on n'empile rien, la taille de la pile utilisée reste constante (si l'interprète est bien fait), au lieu de croître linéairement avec les versions récursives précédentes.

Ce qui nous ramène au premier algorithme, avec la correspondance entre `debut` et `i` pour les indices, et `total` avec `nb` pour les compteurs.

4 Exercice 2 : rendu de monnaie

4.1 La question

Un bout de programme Python à trous, à compléter

```
Pieces = [100,50,20,10,5,2,1]

def rendu_glouton(arendre, solution=[], i=0):
```

```
    if arendre == 0:
        return ...                                  # 1
    p = pieces[i]                                   # BUG
    if p <= ... :                                   # 2
        solution.append(...)                        # 3
        return rendu_glouton(arendre - p, solution, i)
    else :
        return rendu_glouton(arendre, solution, ...)  # 4
```

comportement attendu

```
>>>rendu_glouton_r(68,[],0)
[50, 10, 5, 2, 1]
>>>rendu_glouton_r(291,[],0)
[100, 100, 50, 20, 20, 1]
```

Critiques :

- Erreur dans le sujet. Le nom de la variable globale `Pieces` commence par une majuscule, il est utilisé sans.
- Les concepteurs du sujet ont du mal à respecter les conventions de programmation, qui disent que les morceaux d'un nom (de variable, de méthode) sont séparés par des "underscores". Donc `a_rendre`, et pas `arendre`.
- Et le nom d'une constante globale (les pièces) devrait être en majuscule (`PIECES`).

À ces considérations syntaxiques, dont les rédacteurs du sujet semblent ignorer autant l'existence https://www.python.org/dev/peps/pep-0008 que l'importance (y compris pédagogique), s'ajoute

- Manque de cohérence : si on définit des valeurs par défaut aux paramètres, c'est pour ne pas avoir à les indiquer lors des appels. Les exemples devraient être

```
>>>rendu_glouton_r(68)
[50, 10, 5, 2, 1]
>>>rendu_glouton_r(291)
[100, 100, 50, 20, 20, 1]
```

4.2 Résolution

Il n'y a jamais que 4 trous à compléter.

Quelques observations

- l'exécution des exemples montre qu'il faut retourner une liste. Ça servira sûrement pour le trou numéro 1.
- au cours des appels, le second paramètre est une liste vide au départ (valeur par défaut) qui se remplit (trou 3).
- le troisième paramètre `i` est un indice dans le tableau `Pieces` (ligne du bug, initialement la première. Il faudra bien le faire avancer un jour (trou 4).

Il parait donc raisonnable

- que quand on n'a plus de monnaie à rendre (`arendre == 0`), on retourne la solution qu'on a construite ; `return solution # 1` ;
- qu'on compare la valeur d'une pièce avec la somme à rendre `if p <= arendre:  #2` pour savoir si il faut rendre cette pièce ou pas ;
- que si oui, on ajoute la pièce dans la solution `solution.append(p) # 3`.
- et que sinon, on rende la monnaie avec des pièces de valeurs plus faibles, qui sont plus loin dans la liste `return rendu_glouton(arendre, solution, i + 1) # 4`

4.3 Corrigé

```
PIECES = [100, 50, 20, 10, 5, 2, 1]

def rendu_glouton(a_rendre, solution = [], i = 0):
    if a_rendre == 0:
        return solution                                    # 1
    p = PIECES[i]
    if p <= a_rendre:                                      # 2
        solution.append(p)                                 # 3
        return rendu_glouton(a_rendre - p, solution, i)
    else :
        return rendu_glouton(a_rendre, solution, i + 1)    # 4
```

4.4 Retour à une solution itérative

Dans le code ci-dessus les appels à `rendu_glouton` sont terminaux (on ne fait rien après).

Ce qui veut dire qu'on peut convertir ce code sous forme itérative (avec des boucle).

Un appel terminal signifie : recommencer avec les valeurs indiquée en paramètres. En détail

- `rendu_glouton(a_rendre - p, solution, i)` : diminuer la somme à rendre, et recommencer avec la même pièce
- `rendu_glouton(a_rendre, solution, i + 1)` : recommencer avec la pièce suivante

D'où l'idée d'avoir deux boucles imbriquées

- la boucle extérieure passe en revue les pièces
- la boucle intérieure sur l'utilisation d'une pièce autant que possible pour diminuer la somme à rendre

Le paramètre tampon `solution` se transforme en variable locale de la fonction, retournée à la sortie de la fonction

Une solution simple :

```
PIECES = [100,50,20,10,5,2,1]

def rendu_iteratif(a_rendre):
    solution = []
```

```
    for piece in PIECES:
        while a_rendre >= piece:
            solution.append(piece)
            a_rendre = a_rendre - piece
    return solution

print ("Tests :")
assert rendu_iteratif(68) ==  [50, 10, 5, 2, 1]
assert rendu_iteratif(291) == [100, 100, 50, 20, 20, 1]
print ("Ok !")
```

Que l'on peut accélérer légèrement en retournant dès qu'une pièce a suffi pour épuiser la somme à rendre

```
def rendu_iteratif(a_rendre):
    solution = []
    for piece in PIECES:
        while a_rendre >= piece:
            solution.append(piece)
            a_rendre = a_rendre - piece
        if a_rendre == 0:
            return solution
    # on ne passe jamais ici
```

Dans cette version, on ne sort jamais "par la fin" de la boucle `for`. Le fait que

- la somme à rendre est un entier positif ou nul,
- la valeur 1 est dans `PIECES`,

permet de conclure qu'il arrivera nécessairement que la condition `arendre == 0` soit vraie, au plus tard en rencontrant la pièce 1. Et donc le `return solution` sera forcément exécuté, qui fera sortir "sauvagement" de la boucle `for` et de la fonction.