Algorithmes et Pensée Computationnelle

Structure de données, Itération et récursivité - Exercices Avancés

Au terme de cette série d'exercices avancés, l'étudiant sera capable d'écrire un programme de façon simplifiée en utilisant les notions d'itération et de récursivité.

Cette feuille d'exercices avancés vous permettra d'approfondir vos connaissances des notions vues en cours. Le code présenté dans les énoncés se trouve sur Moodle, dans le dossier "Exercices > Code". Le temps mentionné (🕓) est à titre indicatif.

1 Itération

Quelques rappels de concepts théoriques :

- L'itération désigne l'action de répéter un processus (généralement à l'aide d'une boucle) jusqu'à ce qu'une condition particulière soit remplie.
- Il y a deux types de boucles qui sont majoritairement utilisées :

  Boucle for : Une boucle `for` permet d'itérer sur un ensemble. Cet ensemble peut être une liste, un dictionnaire, une collection, ... La syntaxe d'une boucle en Python est la suivante `for` **nom de variable** `in` suivi du nom de l'élément sur lequel vous voulez itérer. La variable prendra la valeur de chaque élément dans la liste, un par un.

  La syntaxe en java est la suivante : `for` **(type nom_de_variable; condition de fin de boucle; incrémentation à chaque itération)** suivi d'une accolade.

  Boucle while : Les boucles `while` sont des boucles qui s'exécutent de façon continue jusqu'à ce qu'une condition soit remplie. La syntaxe est la suivante : En Python, on écrit d'abord `while`, suivi de la condition à atteindre.

  En Java, il n'y a pas de différence majeure à part le fait qu'il faille mettre entre parenthèses la condition et les deux points sont remplacés par des accolades.

- La fonction `range(n)` permet de créer une liste de nombres de 0 à la valeur passée en argument moins 1 (n-1). Lorsqu'elle est combinée à une boucle `for`, on peut itérer sur une liste de nombres de 0 à n-1.

- Si vous avez fait une erreur dans votre code, il se peut que la boucle `while` ne remplisse jamais la condition lui permettant de sortir. On parle dans ce cas d'une **boucle infinie**. Ceci peut entraîner un crash de votre programme, voire même de votre ordinateur. Il faut toujours s'assurer qu'il y' ait une condition valide dans une boucle `while`.

Question 1: (🕓 *10 minutes*) **Boucle while et des étudiants**
Considérons une liste d'étudiant·e·s contenant le nom de l'étudiant·e suivi d'une valeur booléenne indiquant si l'étudiant·e est présent·e en classe ou non.

l = ["Schmitt", True, "Irma", False, "Khalif", True, "Yasser", False, "Wang", True]

À partir de cette liste et du type de boucle de votre choix, créez un dictionnaire ayant pour clés les noms et pour valeur un booléen indiquant la présence au cours.

Conseil

- **i % 2** : renvoie le reste de i par la division euclidienne de 2. (**i % 2 == 0**) Indique si **i** est pair.
- Si vous utilisez la boucle `for` : Utiliser les indices pour accéder aux éléments de la liste et éviter d'itérer manuellement sur les éléments de la liste.
- Si vous utilisez la boucle `while` : La condition de sortie devrait être en rapport avec la longueur de la liste.

Solution

Solution avec la boucle `for` :

```
l = ["Schmitt", True, "Irma", False, "Khalif", True, "Yasser", False, "Wang", True]
dict_final = {}
for i in range(len(l)):            # i va de 0 à 9
    if i % 2 == 0:                 # Test si i est pair. Si oui, on crée une nouvelle entrée dans le dictionnaire
        dict_final[l[i]] = l[i + 1]
```

Solution avec la boucle `while` :

```
l = ["Schmitt", True, "Irma", False, "Khalif", True, "Yasser", False, "Wang", True]
dict_final = {}
i = 0                              # Variable qui permettra d'itérer et de gérer la boucle while.
while i < len(l):                  # La boucle s'arrêtera donc au bout de 10 itérations. La valeur maximale de i sera 9.
    if i % 2 == 0:
        dict_final[l[i]] = l[i + 1]
    i +=1                          # On incrémente i
```

Pour ce problème, la boucle `for` est plus simple à mettre en place et évite le danger d'une boucle infinie. La boucle `while` est utilisée pour des problèmes plus précis qui ne peuvent pas être résolus par une boucle `for`. La boucle `for` offre aussi l'avantage d'être facile à lire. Pour des problèmes de nécessitant pas un très grand nombre d'itérations, utilisez de préférence une boucle `for`. Par contre, si vous l'utilisez pour trouver un élément dans un grand ensemble (par exemple, une liste contenant un grand nombre d'enregistrements), utilisez une boucle `while` car elle offre l'avantage de sortir de la boucle une fois que la condition initiale est fausse sans avoir à parcourir l'ensemble des éléments (par exemple, votre boucle s'arrêtera lorsque vous aurez trouvé l'élément que vous recherchez dans la liste volumineuse).

Question 2: (*5 minutes*) **Différence entre Itération et Récursivité**
Quel est la différence dans le traitement de problèmes entre l'itération et la récursivité ?

Conseil

— L'itération traite les problèmes de manière "contrôlée". En effet, lors d'un appel itératif, la boucle est répétée tant que la condition est vérifiée.
— La récursivité résout les problèmes en résolvant des sous-problèmes de ceux-ci.

Solution

— L'itération va résoudre de manière plus "intuitive" les problèmes, c'est-à-dire répéter certaines actions qui permettent de résoudre un problème souvent au prix de l'efficience.
— La récursivité permet de diviser un problème en sous-problèmes et ainsi de suite jusqu'au moment où le sous-problème est soluble et permet de résoudre tous les problèmes jusqu'à celui original. La récursivité cache une faille, un appel à elle même infini. De manière semblable aux boucles `while`, la récursivité peut causer le crash du programme voir de l'ordinateur. Lorsque vous cherchez à faire une récursivité il faut donc systématiquement poser avant tout la condition finale pour l'arrêt.

2 Récursivité

La récursivité est le fait d'appeler une fonction de façon récursive c'est-à-dire une fonction qui s'appelle elle-même de façon directe ou indirecte.

Question 3: (*5 minutes*) **Lecture de code - 3** Optionnel
Qu'affiche le programme suivant ?

```
def fct_a(mot) :

if len(mot) == 1 :
    if mot[0] == "a" :
        return 1
    else :
        return 0
else :
    if mot[0] == "a" :
        return 1 + fct_a(mot[1:])
    else :
        return 0 + fct_a(mot[1:])

print(fct_a("blablabla"))
```

Choisissez parmi les possibilités suivantes :

— 1

— 2

— 3

— 4

A quoi sert la fonction **fct_a(mot)** ?

Conseil

Aidez vous d'un papier et effectuez les étapes une à une afin de bien comprendre comment le code est articulé.

Solution

3

Cette fonction compte le nombre de "a" contenus dans un mot que vous lui passez.

Question 4: (*10 minutes*) **Fonction factoriel()**
Ecrivez la fonction **factoriel()** suivant une approche récursive.

Pour rappel, cette fonction prend un entier et retourne le factoriel de ce dernier tel que :

factoriel(1) = 1, factoriel(2) = $1 * 2 = 2$, factoriel(3) = $1 * 2 * 3 = 6$, factoriel(4) = $1 * 2 * 3 * 4 = 24$,...

Conseil

On peut écrire cette fonction comme suit :

f(n) = f(n) * f(n-1) si $n > 0$

f(n) = 1 si n = 0

Solution

Python :

```python
def factoriel(x) :
    if x == 0:
        return 1
    else :
        return x * factoriel(x-1)
```

Java :

```java
public static int factoriel(int x){
    if (x == 0){
        return 1;
    }
    else{
        return x*factoriel(x-1);
    }
}
public static void main(String[] args) {
    System.out.println(factoriel(5));
}
```

Question 5: (*15 minutes*) **Fibonacci - Java**

Ecrivez deux fonctions permettant de calculer un nombre donné de la suite de Fibonacci. L'une doit utiliser la récursivité, et l'autre l'itération. Pour rappel, chaque élément de la suite de Fibonacci est la somme des deux derniers éléments. L'élément 0 vaut 0, l'élément 1 vaut 1 et l'élément 2 vaut 1.

Début de la suite : [0,1,1,2,3,5,8,13,...]

Conseil

L'algorithme permettant de faire une suite de Fibonacci a été présenté dans les diapositives du cours. En cas de difficulté, n'hésitez pas à vous y référer.

Solution

Via Itération :

```
1   public static int fibonacci_i(int n){
2       if (n==0 || n==1){
3           return n;
4       }
5       else{
6           int old_fib = 1;
7           int new_fib = 1;
8           int tmp;
9           for (int i=2; i<n; i++){
10              tmp = new_fib;
11              new_fib = new_fib+old_fib;
12              old_fib = tmp;
13          }
14          return new_fib;
15      }
16  }
```

Via récursivité :

```
1   public static int fibonacci_r(int n){
2       if (n==0 || n==1){
3           return n;
4       }
5       else{
6           return fibonacci_r(n-1) + fibonacci_r(n-2);
7       }
8   }
```