IFT-1004 - Introduction à la programmation

Module 10 : Récursivité

Honoré Hounwanou, ing.

Département d'informatique et de génie logiciel
Université Laval

Table des matières

Les algorithmes récursifs

Les problèmes définis récursivement

Les structures de données récursives

Lectures et travaux dirigés

Objectif

S'introduire à un concept fondamental en programmation, **la récursivité**, en solutionnant des problèmes de manière récursive, puis en explorant certains problèmes et certaines structures de données qui sont définis récursivement.

Les algorithmes récursifs

Introduction

Plusieurs problèmes sont résolus de manière récursive parce qu'ils sont naturellement décrits de façon récursive (induction).

En mathématiques entre autres : calcul de la **factorielle**, la suite de **Fibonacci**, le problème du **plus grand commun diviseur** (PGCD), etc. La résolution du problème est simplifiée dans ces cas.

Diviser pour régner : ré-appliquer un même traitement sur un échantillon de données d'une taille de plus en plus petite.

En informatique, la programmation avancée utilise souvent des techniques de programmation récursive.

Définition

Soit *f* une fonction comprenant un appel à elle-même, soit *directement*, soit *indirectement*. Alors, *f* est une fonction **récursive**.

Récursivité directe : une fonction comporte, dans sa définition, au moins un appel à elle-même.

```
def f_1(...):
    ...
    x = f_1(...)
    ...
    return x
```

Structure générale

```
def fonction(...):
    # Vérification des conditions de convergence
    if ...:
        raise Exception(...)

    # Vérification des conditions d'arrêt
    if ...:
        return ...

    # Appel(s) récursif(s)
    x = fonction(...)

    # Retour
    return x
```

Diviser pour régner

Les algorithmes de type **diviser pour régner** divisent le problème en un sous-problème du même type (résolu par le même algorithme), jusqu'à ce que le problème soit si simple qu'on en connaît déjà la solution.

Diviser pour régner : la somme d'une liste

Idée : Pour calculer la somme des éléments d'une liste de n éléments, il s'agit d'additionner **le premier élément** à **la somme des** $n-1$ **éléments du reste de la liste**.

Récursion : $f(x[0], x[1], \ldots, x[n-1]) = x[0] + f(x[1], \ldots, x[n-1])$

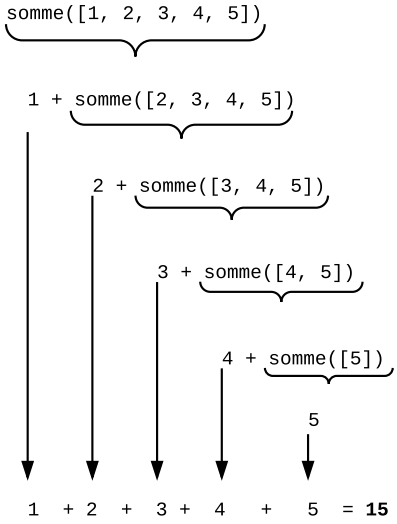
Condition d'arrêt : $f(x[0]) = x[0]$ (un seul terme à additionner)

Convergence : Pour tout $n > 1$, n décroît de 1 à chaque appel récursif. On a donc convergence pour $n \geq 0$.

Diviser pour régner : la somme d'une liste

```
def somme(liste):

    # Conditions d'arrêt
    if len(liste) == 0:
        return 0

    if len(liste) == 1:
        return liste[0]

    # Appel récursif et retour
    return liste[0] + somme(liste[1:])
```

La somme d'une liste : trace des appels



La somme d'une liste : version non récursive

```
def somme(liste):
    la_somme = 0
    for element in liste:
        la_somme += element

    return la_somme
```

Diviser pour régner : le palindrome

Un palindrome est un mot qui peut être lu de la même manière de gauche à droite ou de droite à gauche.

Idée : Un mot est un palindrome si **son premier et son dernier caractères sont les mêmes**, et si **la partie « à l'intérieur » du mot est un palindrome**.

Diviser pour régner : le palindrome

Récursion : `chaine` est un palindrome si :

- `chaine[0] == chaine[-1]`
- `chaine[1:-1]` est un palindrome

Conditions d'arrêt :

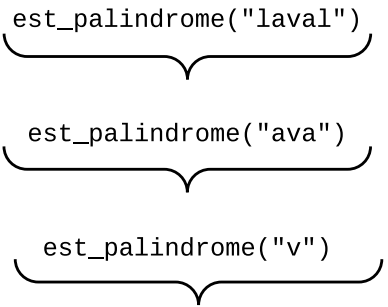
1. la chaîne n'a aucun caractère → VRAI
2. la chaîne a un seul caractère → VRAI
3. Si `chaine[0] != chaine[-1]` → FAUX

Convergence : Une chaîne a nécessairement une longueur ≥ 0. Sa longueur diminue de 2 à chaque appel récursif pour `len(chaine)` ≥ 2. On a donc convergence pour `len(chaine)` ≥ 0.

Diviser pour régner : le palindrome

```python
def est_palindrome(chaine):

    # Conditions d'arrêt
    if len(chaine) <= 1:
        return True

    if chaine[0] != chaine[-1]:
        return False

    # Appel récursif et retour
    return est_palindrome(chaine[1:-1])
```

Le palindrome : trace des appels



Le palindrome : version non récursive

```
def est_palindrome(chaine):

    # Conditions d'arrêt
    if len(chaine) <= 1:
        return True

    for i in range(len(chaine) // 2):
        if chaine[i] != chaine[-i-1]:
            return False

return True
```

Diviser pour régner : la factorielle

La factorielle d'un nombre n, nommée $n!$, est définie par le produit $1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$

Idée : La factorielle d'un nombre n est le produit entre n et la factorielle du nombre $n - 1$.

Récursion : $f(n) = n \cdot f(n - 1)$

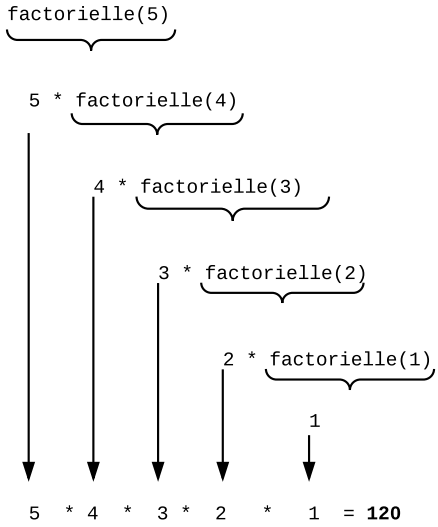
Conditions d'arrêt : Par définition, $0! = 1! = 1$.

Convergence : Pour tout $n \geq 2$, la soustraction par 1 nous amènera vers la condition d'arrêt. On a donc convergence pour $n \geq 0$.

Diviser pour régner : la factorielle

```
def factorielle(n):
    # Condition de convergence:
    if n < 0:
        raise ValueError("Non convergence")

    # Conditions d'arrêt:
    if n == 0 or n ==1:
        return 1

    # Appel récursif et retour
    return n * factorielle(n - 1)
```

Le factorielle : trace des appels



La factorielle : version non récursive

```
def factorielle(n):
    # Condition de convergence:
    if n < 0:
        raise ValueError("Non convergence")

    resultat = 1
    for i in range(2, n+1):
        resultat *= i

    return resultat
```

Les problèmes définis récursivement

La suite de Fibonacci

La suite de Fibonacci est une suite de nombres très étudiée en mathématiques. Elle a été nommée en l'honneur de Leonardo Bonacci (aussi connu sous plusieurs autres noms dont Leonardo Pisano). Chaque terme de cette suite (sauf les deux premiers) sont obtenus en additionnant les deux termes précédents : $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.

Récursion : $f(n) = f(n-1) + f(n-2)$

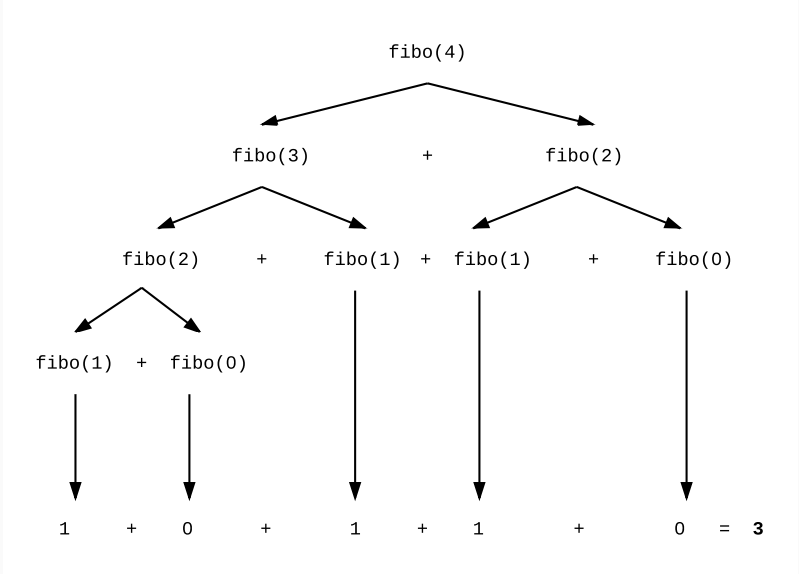
Conditions d'arrêt : $f(0) = 0, f(1) = 1$

Convergence : Pour tout $n \geq 2$, nous faisons deux appels récursifs avec de plus petites valeurs de n, jusqu'à ce que n soit égal à 0 ou 1. On a donc convergence pour $n \geq 0$.

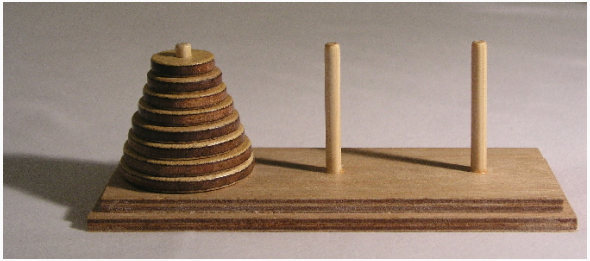
La suite de Fibonacci

```
def fibonacci(n):
    # Condition de convergence:
    if n < 0:
        raise ValueError("Non convergence")

    # Conditions d'arrêt:
    if n == 0 or n == 1:
        return n

    # Appel récursif et retour
    return fibonacci(n - 2) + fibonacci(n - 1)
```

Le suite de Fibonacci : trace des appels



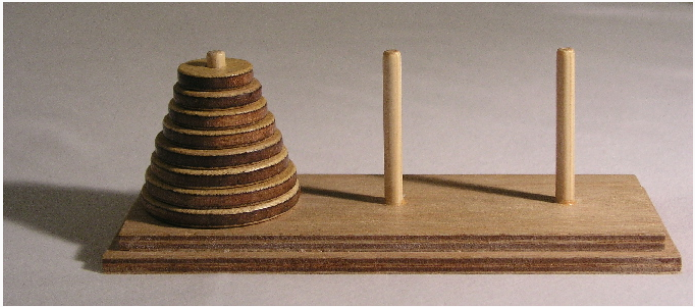
Les tours de Hanoï



Le jeu mathématique des tours de Hanoï consiste à déplacer une tour de disques d'une pôle vers une autre pôle, en ayant accès à une pôle intermédiaire. Les règles sont les suivantes :

- On ne peut transférer qu'un seul disque à la fois
- Les disques ont des diamètres différents
- À aucun moment, un disque de diamètre supérieur ne peut reposer sur un disque de diamètre inférieur.

Les tours de Hanoï



Pour un certain nombre de disques n, le problème est soluble si nous sommes capables d'accomplir les actions suivantes :

1. Déplacer **récursivement** $n - 1$ disques de la tour source vers la tour intermédiaire
2. Déplacer le disque restant de la tour source vers la tour de destination
3. Déplacer les $n - 1$ disques de la tour intermédiaire vers la tour de destination

Les tours de Hanoï

```
def hanoi(n, source, auxiliaire, destination):
    if n <= 0:
        raise ValueError("Erreur!")

    if n == 1:
        print("Déplacer {} vers {}".format(source, destination))

    else:
        hanoi(n - 1, source, destination, auxiliaire)
        print("Déplacer {} vers {}".format(source, destination))
        hanoi(n - 1, auxiliaire, source, destination)
```

Les structures de données récursives

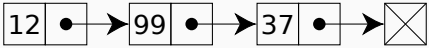
Les structures de données récursives

Soit *o* **un objet comprenant une instance de lui-même**. Alors *o* est une *structure récursive*.

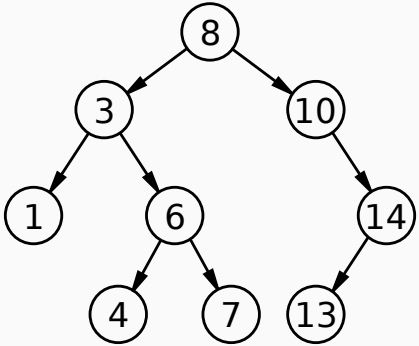
```
class Tata:
    def __init__(self, n):
        self.n = n
        if n != 0:
            self.suivant = Tata(n - 1)
        else:
            self.suivant = None
```

Les structures de données récursives

Liste chaînée :



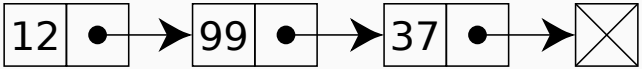
Arbre binaire (de recherche) :



Les listes chaînées en python

```
class Noeud:
    def __init__(self, donnee):
        self.donnee = donnee
        self.suivant = None


class ListeChainee:
    def __init__(self):
        self.noeud_courant = None

    def inserer(self, donnee):
        nouveau_noeud = Noeud(donnee)
        nouveau_noeud.suivant = self.noeud_courant
        self.noeud_courant = nouveau_noeud
```

Les listes chaînées en python




```
lc = Listechainee()
lc.inserer(37)
lc.inserer(99)
lc.inserer(12)

print(lc.noeud_courant.donnee)
print(lc.noeud_courant.suivant.donnee)
print(lc.noeud_courant.suivant.suivant.donnee)
```

Les arbres binaires en python

```
class NoeudArbreBinaire:
    def __init__(self, donnee):
        self.donnee = donnee
        self.gauche = None
        self.droite = None


class ArbreBinaireDeRecherche:
    def __init__(self):
        self.racine = None

    def inserer(self, donnee):
        self.racine = self.__inserer_recursif(self.racine,  donnee)

    [...]
```

Les arbres binaires en python

```
    [...]
    def __inserer_recursif(self, racine, donnee):
        if racine is None:
            racine = NoeudArbreBinaire(donnee)
        else:
            if donnee <= racine.donnee:
                racine.gauche = self.__inserer_recursif(racine.gauche, donnee)
            else:
                racine.droite = self.__inserer_recursif(racine.droite, donnee)
    return racine

if __name__ == '__main__':
    a = ArbreBinaireDeRecherche()
    a.inserer(8)
    a.inserer(3)
    a.inserer(6)
    print(a.racine.gauche.droite.donnee)
```

La récursivité en bref

- La récursion n'est pas une panacée (remède universel) : si on peut aisément la remplacer par une boucle, il vaut mieux le faire.
- Pour certains problèmes, une solution récursive est plus élégante et plus simple à comprendre qu'une solution itérative. Pour d'autres problèmes, c'est l'inverse.
- Une application habile de la récursivité permet parfois un gain en performance, mais attention : ce n'est pas le cas en général, car appeler récursivement une fonction implique l'ajout de plusieurs étapes qui ralentissent l'exécution
- La récursion permet également de traiter des structures arbitraires (listes de listes, documents *XML*, etc.), ce qui est beaucoup plus complexe avec des boucles imbriquées, car on ne peut pas prédire combien de boucles imbriquées seraient nécessaires

La récursivité en bref

Exemple : Supposons que l'on veut récursivement sommer les éléments d'une liste de listes de listes, etc...

```
def somme(liste):
    resultat = 0

    for element in liste:
        if isinstance(element, list):
            resultat += somme(liste)
        else:
            resultat += element

    return resultat
```

Lectures et travaux dirigés

Lectures et travaux dirigés

- Aucune lecture
- Travaux dirigés : Exercices sur la récursivité

Questions ?