

Chapitre 1: Introduction à l'algorithmique

Brice Mayag
brice.mayag@dauphine.fr

M1 MSI

Sommaire

- 1 Présentation du cours
- 2 Introduction et définitions
 - Pourquoi l'étude des algorithmes ?
 - Définitions
- 3 Paradigmes et langages de programmation
- 4 Fondements des langages : la récursivité
 - Algorithmes récursifs
- 5 Le langage Python

Sommaire

- 1 Présentation du cours
- 2 Introduction et définitions
 - Pourquoi l'étude des algorithmes ?
 - Définitions
- 3 Paradigmes et langages de programmation
- 4 Fondements des langages : la récursivité
 - Algorithmes récursifs
- 5 Le langage Python

Programme du cours

Les grandes notions

- Les bases de l'algorithmique
- Structures de données basiques et avancées : listes, files, arbres de recherche
- Application au filtrage collaboratif

Évaluation

- En continue durant les séances de cours

Sommaire

- 1 Présentation du cours
- 2 Introduction et définitions
 - Pourquoi l'étude des algorithmes ?
 - Définitions
- 3 Paradigmes et langages de programmation
- 4 Fondements des langages : la récursivité
 - Algorithmes récursifs
- 5 Le langage Python

L'algorithmique ?

Définition (informelle)

Un algorithme est la composition d'un ensemble fini d'étapes, chaque étape étant formée d'un nombre fini d'opérations dont chacune est :

- définie de façon rigoureuse et non ambiguë ;
- effective (i.e. pouvant être réalisée en un temps fini).

La notion d'algorithme est plus générale que celle de programme (indépendant du langage de programmation utilisé).

Un peu d'histoire

Le mot algorithme vient du nom du mathématicien Al Khwarizmi (820) : introduction de la numérotation décimale et des calculs s'y rapportant.

Des besoins contradictoires

Un algorithme doit :

- être simple à comprendre, à mettre en oeuvre et à mettre au point ;
- mettre intelligemment à contribution les ressources de l'ordinateur, et plus précisément, il doit s'exécuter rapidement.

Un algorithme : c'est quoi ?

Le crumble aux pommes du chat qui tousse

Un problème

Très facile

1

Ingrédients (pour 6 personnes) :

- 6 belles pommes (des Canada par exemple)
- 250 g de cassonade
- 150 g de farine
- 125 g de beurre (le sortir 1/2 heure avant de commencer la recette)
- le jus d'un citron
- une petite cuillère de cannelle en poudre
- 1 sachet de sucre vanillé

Des données

Préparation :

Préchauffer le four à thermostat 7 (210°C).

Peler, évider et découper les pommes en cubes grossiers, les répartir dans un plat allant au four, verser dessus le jus du citron, la cannelle et le sucre vanillé.

Dans un saladier, mélanger la farine et la cassonade.
Puis ajouter le beurre en petits cubes et mélanger à la main de façon à former une pâte grumeleuse.

Emietter cette pâte au dessus des pommes de façon à les recouvrir.
Mettre au four une bonne 1/2 heure.

Servir tiède avec de la crème fouettée ou de la glace à la vanille.

Des instructions



Un résultat

Classiquement assimilable à une recette de cuisine

Algorithme : définitions

Un algorithme =

- Description précise des opérations à faire pour résoudre un problème (suite d'instructions).
- Procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs et qui donne en sortie une valeur, ou un ensemble de valeurs.

Un *bon* algorithme =

- Un algorithme **correct** : i.e. pour chaque instance en entrée, l'algorithme se termine en produisant la bonne sortie
⇒ Savoir prouver un algorithme
- Un algorithme **efficace** : mesure de la durée que met un algorithme pour produire un résultat
⇒ Savoir analyser la complexité d'un algorithme : i.e. détermination de l'espace mémoire et du temps d'exécution nécessaire à la résolution du problème.

Pour un problème donné, **plusieurs** algorithmes ou **aucun** sont possibles.
Un algorithme se termine en un **temps fini**.

Un exemple simple : tri par sélection

Problème de tri

- Données : une suite de n nombres, $(e_0, e_1, \dots, e_{n-1})$
- Résultat : permutation de la suite donnée en entrée $(e'_0, e'_1, \dots, e'_{n-1})$ de telle sorte que $e'_0 \leq e'_1 \leq \dots \leq e'_{n-1}$

Principe du tri par sélection

- Rechercher le plus grand élément (ou le plus petit), le placer en fin de tableau (ou en début).
- Recommencer avec le second plus grand (ou le second plus petit), le placer en avant-dernière position (ou en seconde position) et ainsi de suite jusqu'à avoir parcouru la totalité du tableau

Un exemple simple : tri par sélection

Principe

On veut trier, du plus petit au plus grand, la liste :

3	2	7	1	5
---	---	---	---	---

- 1 L'élément le plus petit se trouve en 3ème position (si on commence à compter à partir de zéro) :

3	2	7	1	5
---	---	---	---	---

- 2 On échange l'élément le plus petit (en 3ème position) avec le premier :

1	2	7	3	5
---	---	---	---	---

- 3 On continue en considérant le même tableau, et en ignorant son premier élément (qui est déjà trié)

1	2	7	3	5
---	---	---	---	---

- 5 Et ainsi de suite, en ignorant à chaque fois les éléments déjà triés (en rouge).

Analyse d'un algorithme

- Simplicité et intelligibilité de l'algorithme
- Efficacité de l'algorithme :
 - Temps d'exécution
 - Occupation de la mémoire
 - Quantité de trafic généré sur un réseau
 - Quantité de données déplacées sur le disque
 - ...

Sommaire

- 1 Présentation du cours
- 2 Introduction et définitions
 - Pourquoi l'étude des algorithmes ?
 - Définitions
- 3 Paradigmes et langages de programmation
- 4 Fondements des langages : la récursivité
 - Algorithmes récursifs
- 5 Le langage Python

Langages de programmation

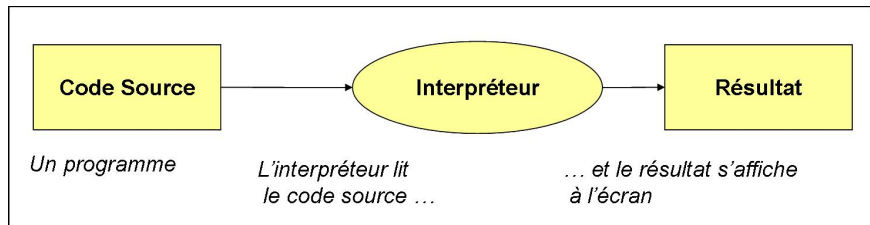
Ensembles de mots clefs et de règles pour former des phrases pouvant être traduites par la machine (binaire).

Plusieurs niveaux

- Langages de bas niveau : instructions élémentaires très *proches de la machine* (ex : Assembleur : `V AX, [0110]` signifie *copier le contenu de 0110 dans le registre AX* ; langage machine)
- Langages de haut niveau : instructions plus abstraites (C, C++, Perl, Java, Python).

Production de programmes : Interprétation

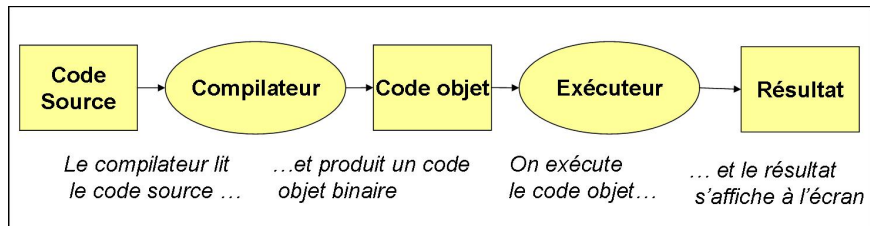
Interprétation



Chaque ligne du code source est traduite au fur et à mesure en instructions qui sont directement exécutées, i.e. l'interpréteur est utilisé à chaque exécution.

Production de programmes : Compilation

Compilation



Compilation = traduction du code écrit dans un langage dit source en langage objet ou cible (analyse lexicale, syntaxique, sémantique et production du code objet). L'édition de liens ..

Sommaire

- 1 Présentation du cours
- 2 Introduction et définitions
 - Pourquoi l'étude des algorithmes ?
 - Définitions
- 3 Paradigmes et langages de programmation
- 4 Fondements des langages : la récursivité
 - Algorithmes récursifs
- 5 Le langage Python

Algorithmes récursifs

Définition

Un algorithme est dit **récursif** s'il s'appelle lui même.

Récusivité

Principe qui consiste à décrire les étapes nécessaires à la résolution de problèmes en utilisant la résolution du même problème sur des entrées plus petites.

Algorithmes récursifs : exemple

Factorielle d'un nombre

On peut définir la factorielle d'un nombre de la manière suivante :

$$n! = \begin{cases} 1 & \text{si } n \leq 1 \\ n(n-1)! & \text{sinon} \end{cases}$$

Algorithmes récursifs : exemple

Factorielle : algorithme itératif

Algorithm 1 FACTORIEL-ITERATIF (n : entier positif)

```
1:  $x \leftarrow 1$ 
2: for  $i \leftarrow 1$  to  $n$  do
3:    $x \leftarrow i * x$ 
4: end for
5: return  $x$ 
```

Algorithmes récursifs : exemple

Factorielle : algorithme récursif

Algorithm 2 FACTORIEL-RECURSIF (n : entier positif)

```
1: if  $n = 1$  then  
2:   return 1  
3: else  
4:   return  $n$  FACTORIEL-RECURSIF( $n - 1$ )  
5: end if
```

Sommaire

- 1 Présentation du cours
- 2 Introduction et définitions
 - Pourquoi l'étude des algorithmes ?
 - Définitions
- 3 Paradigmes et langages de programmation
- 4 Fondements des langages : la récursivité
 - Algorithmes récursifs
- 5 Le langage Python

Le langage Python

C'est

- Un langage de scripts (de petits programmes très simples chargés d'une mission très précise sur votre ordinateur) ;
- Un langage impératif interprété (c'est-à-dire que les instructions que vous lui envoyez sont "transcrites" en langage machine au fur et à mesure de leur lecture) contrairement à un langage compilé (C/C++)

Le langage Python

Python est un langage de programmation

Ce qui définit un langage de programmation

- La façon de représenter **symboliquement** les **structures de données**.
- La façon de gérer le **contrôle** des programmes (que faire et dans quel ordre)

Comme tout langage de programmation, Python fournit un certain nombre de types de données prédéfinis (booléens, entiers, flottants, chaîne de caractères, listes) et de structure de contrôle (les boucles for, while et les conditions if_then_else_).

Les structures de données

Structure de données

Une **structure de données** est définie par 3 choses :

- 1 Un **type** qui est simplement un nom permettant de classifier les données relevant de ce nom.
- 2 Un **ensemble** qui définit avec précision les éléments appartenant au type.
- 3 des **opérations** utilisées par les programmes pour calculer sur ces données

On parle de **structures algébriques**.

Les booléens

Ce type de données est prédéfini dans Python par :

- son nom `bool`
- ses valeurs `{True, False}`
- ses opérations
 - `not` : $bool \rightarrow bool$,
 - `and` : $bool \times bool \rightarrow bool$, et
 - `or` : $bool \times bool \rightarrow bool$

Nom vient de George Boole [1815-1864] mathématicien anglais intéressé par les propriétés algébriques des valeurs de vérité.

Les entiers relatifs

Ce type de données est prédéfini dans Python par :

- son nom `int`
- son ensemble de valeurs est théoriquement \mathbb{Z} mais est en fait borné selon la machine utilisée.
- ses opérations
 - `+, -, *, /, %, **` : $int \times int \rightarrow int$,
 - `==, >, <, >=, <=, !=` : $int \times int \rightarrow bool$

Les opérations de calcul (`+, -, ** ...`) sont prioritaires sur les opérateurs de comparaison (`==, <=, ...`).

Quelques exemples d'utilisation des entiers

```

>>> type(46) #type : fonction pr'\{e}d\{e}finie fournissant
type 'int'          #le type d'une expression
>>> print 5 * 7    #print : fonction pr'\{e}d\{e}finie d'impression
35
>>> 5 / 2        # / quotient de la division euclidienne
2
>>> 5 % 2 # % reste de la division euclidienne
1
>>> 5 ** 3 # fonction d'exponentation
125
>>> 5 < 3 # pr'\{e}dicat de comparaison "strictement inf\{e}rieur"
False
>>> 5 == 3 + 2 # op'\{e}ration + est prioritaire sur ==
True
>>> 5 = 3 + 2
SyntaxError: can't assign to literal

```

Les nombres réels

En fait, les nombres réels ne sont pas représentables en machine. On les approxime par des nombres rationnels (i.e. avec des virgules).

Ce type de données est prédéfini dans Python par :

- son nom `float`
- son ensemble de valeurs est théoriquement \mathbb{R} mais est en fait un sous-ensemble de \mathbb{Q} . De plus, il est borné selon la machine utilisée.
- ses opérations
 - `+`, `-`, `*`, `/`, `**` : `float` \times `float` \rightarrow `float`,
 - `int` : `float` \rightarrow `int`, et `float` : `int` \rightarrow `float`
 - `==`, `>`, `<`, `>=`, `<=`, `!=` : `float` \times `float` \rightarrow `bool`

Les flottants s'écrivent en utilisant le point pour séparer la partie décimale de la partie entière.

Quelques exemples d'utilisation des nombres réels

```
>>> type(46.8)
<type 'float'>
>>> 5.0 / 2. # le point distingue un flottant d'un entier,
2.5
>>> 5. / 2 # 2 est converti en flottant
2.5
>>> 7.5 * 2
15.0
>>> int (7.5 * 2)
15
>>> int (7.75)
7
>>> float (3)
3.0
```

Les chaînes de caractères

Type immuable (on ne peut pas changer les valeurs)

Description formelle

Une **chaîne de caractères** est une application $s : S \rightarrow \text{char}$ où S est une séquence finie d'entiers positifs $\{0, \dots, n\}$ et **char** est l'ensemble des 256 caractères ascii.

Ce type de données est prédéfini dans Python par :

- son nom **str**
- son ensemble de valeurs est tout mot fini (i.e. une suite de caractères ascii).
- ses opérations
 - $+$: $\text{str} \times \text{str} \rightarrow \text{str}$ (concaténation de deux chaînes de caractères),
 - $*$: $\text{str} \times \text{int} \rightarrow \text{str}$ l'entier doit être positif,
 - **len** : $\text{str} \rightarrow \text{int}$ (longueur d'une chaîne de caractères),
 - $==, >, <, >=, <=, !=$: $\text{str} \times \text{str} \rightarrow \text{bool}$

Les chaînes de caractères - suite

En python, une chaîne se décrit en extension en écrivant le mot avec des guillemets : $s = "a_1 \dots a_n"$.

La notation $s[i]$ pour $i = 0, \dots, (n - 1)$ dénote la lettre à la position $i + 1$ dans la chaîne s .

On peut accéder à des sous chaînes de caractères d'une chaîne donnée. Soit s une chaîne de longueur n . $s[i : j]$ avec $i < j$ et $i, j \leq n$, dénote la sous-chaîne $s[i]s[i + 1] \dots s[j - 1]$

Accès aux sous-chaînes par les (intervalles d')index

```
>>> test = '1234567'
>>> test[3:6]
'456'
>>> test[0]
'1'
>>> test[3]
'4'
>>> test[6]
'7'
>>> test[3:4]
'4'
```

Un programme python

Un programme est une suite de définition de fonction.

Fonction python

Une **fonction** python se déclare de la façon suivante

```
def nom_fonction (arg1,...,argn):
    # texte de description de la fonction
    Corps de la fonction
```

Le corps de la fonction se définit inductivement à partir :

- des **affectations de variables** $x = expr$ (x est du type de l'expression $expr$)
- des **conditionnelles**
- des boucles **for** et **while**

Les résultats des fonctions sont retournés à l'environnement appelant par le mot clé **return**. Une fonction sans **return** est appelée une **procédure**.

Les expressions conditionnelles

Condition en python

Une **expression conditionnelle** python se déclare de la façon suivante

```
if expression_conditionnelle :  
    Instructions  
else :  
    Instructions
```

Les expressions conditionnelles - suite

Dans le cas où l'on ne souhaite pas rien faire lorsque la condition est fausse, on peut omettre la partie "else".

Enfin, il arrive que l'on ait des "cascades" d'expressions conditionnelles. On peut alors utiliser le mot clé `elif`.

```
if expression_conditionnelle :  
    Instructions  
elif expression_conditionnelle :  
    Instructions  
else :  
    Instructions
```

Les boucles While et for

Boucle while

Une boucle **while** python se déclare de la façon suivante

```
while expression_conditionnelle :  
    Instructions
```

Boucle for

Une boucle **for** python se déclare de la façon suivante

```
for variable in collection :  
    Instructions
```

Les collections dans les boucles “for” peuvent être de plusieurs formes :

- “range(ind_début,ind_fin,pas)”. La variable va prendre pour valeur à chaque étape de la boucle “for” ind_début,ind_début+pas,...,ind_fin.
- une chaîne de caractères ou une liste. Dans ce cas là, la variable va parcourir toute les valeurs se trouvant aux positions allant de 0 à $len(s) - 1$ où s est la chaîne de caractères ou la liste.

Variables locales et variables globales

Lorsqu'une fonction utilise une variable, elle est toujours "locale" à cette fonction.

```
def triple (n) :
    resultat = n * 3
    return resultat

>>> triple(4)
12
>>> resultat          #variable locale n'est pas connue
Traceback (most recent call last): ...
>>> resultat =128 #variable globale
>>> resultat(2)
Traceback (most recent call last): ....
>>> triple(2)
6
>>> resultat #variables globale inchang\'{e}e m\^{e}me si
                #utilis\'{e} entretemps comme variable locale
128
```

Les listes/tableaux

Description formelle

Une **liste** est une application $s : S \rightarrow TYPE$ où S est une séquences d'entiers positifs $\{0, \dots, n\}$ et **TYPE** est l'union de tous les types de données.

Ce type de données est prédéfini dans Python par :

- son nom **lists**
- son ensemble de valeurs est tout mot fini sur l'ensemble TYPE pris comme alphabet.
- ses opérations
 - $+$: $list \times list \rightarrow list$ (concaténation de deux listes),
 - $*$: $list \times int \rightarrow list$ l'entier doit être positif,
 - **len** : $list \rightarrow int$ (longueur d'une liste),
 - $==, >, <, >=, <=, !=$: $list \times list \rightarrow bool$

En python, une liste se décrit en extension en écrivant le mot entre crochets : $l = [a_1, \dots, a_n]$. La notation $[]$ dénote la liste vide.

Les listes - suite

En python, les listes sont mutables, i.e., on peut changer les valeurs aux différentes positions. On utilise la notation : $l[i] = e$ met l'élément e à la position i dans la liste l

On peut accéder à des sous listes d'une liste donnée. Soit l une liste de longueur n . $l[i:j]$ avec $i < j$ et $i, j \leq n$, dénote la sous-liste $l[i]l[i+1] \dots l[j]$

Convention : $l[:i]$ et $l[i:]$ dénotent les i premiers et les $n - i$ derniers éléments de la liste l , respectivement.

Quelques illustrations sur les listes

```
>>> type([1,2])
<type 'list'>
>>> [2+3,"toto",2.5]
[5, 'toto', 2.5]
>>> type([2+3,"toto",2.5])
<type 'list'>
>>> [5,[2,2],"toto"]
[5, [2, 2], 'toto']
>>> ["toto",5,"tutu",5.5][1]
5
>>> ["toto",5,"tutu",5.5][1:3]
[5, 'tutu']
>>> "tutu" in ["toto",5,"tutu",5.5]
True
>>> "tu" in ["toto",5,"tutu",5.5]
False
```

Un exemple de programme

Recherche d'un élément dans une liste

```
def find (e,l):  
    if l == []:  
        return False  
    else:  
        if e == l[0]:  
            return True  
        else:  
            return find(e,l[1:len(l)])
```