RÉVISIONS D'ALGORITHMIQUE

Chapitre 1

mardi 4 septembre 2018

Lycée Chaptal – PT*

http://www.mickaelprost.fr

Algorithme

> Un *algorithme* est une suite finie d'instructions non ambiguës pouvant être exécutées de façon automatique.

Un algorithme est caractérisé – entre autres – par :

1. son exactitude
2. son efficacité

   complexité temporelle / *complexité en place mémoire*
3. sa simplicité
4. sa qualité

Les données en Python sont typées. On rencontre notamment :

- le type `int` (entier)
  opérations : `+` `-` `*` `**` `//` `%`
- le type `float` (flottant)
- le type `str` (chaîne de caractères)
  concaténation : `+` répétition : `*`
- le type `bool` (booléen)
  opérations : `not` `and` `or` comparaison : `==` `!=` `>=`

```
>>> type([1, 2, 3])
<class 'list'>
>>> type(range(0,4))
<class 'range'>
>>> list(range(0,4))
[0, 1, 2, 3]
>>> import math
>>> type(math.cos)
<class 'builtin_function_or_method'>
```

Une variable est de façon imagée l'association d'un nom et d'une valeur.

AFFECTATION SIMPLE

```
>>> a = 2
>>> type(a)
<class 'int'>
>>> b = a
>>> b
2
>>> a = 3
>>> b
2
```

AFFECTATION MULTIPLE

```
>>> x, y = 2, 5
>>> x
2
>>> y
5
>>> x, y = y, x
>>> x
5
>>> y
2
```

Une fonction est la traduction d'un algorithme qui :

- prend en entrée un ou des arguments ;
- effectue une suite d'instructions ;
- retourne un résultat.

Une fonction peut être *définie par l'utilisateur* :

```
>>> def f(x):
...     ''' Calcul le cube '''
...     return x**3
...
>>> f(2)
8
```

Une fonction est la traduction d'un algorithme qui :

- prend en entrée un ou des arguments ;
- effectue une suite d'instructions ;
- retourne un résultat.

Une fonction peut être *prédéfinie et contenue dans une bibliothèque* :

```
>>> import math
>>> math.sin(0)
0.0
>>> from math import * # bof !
>>> sin(pi)
1.2246467991473532e-16
>>> import math as m # mieux !
>>> m.sin(m.pi)
1.2246467991473532e-16
```

- Une méthode peut être assimilée à une fonction.
- Elle agit sur un objet selon la syntaxe suivante :

`objet.methode(arguments)`

Exemple 1 :

```
>>> L = [1, 4, 2, 9, 7, 2]
>>> L.append(2)
>>> L
[1, 4, 2, 9, 7, 2, 2]
>>> L.count(2)
3
>>> L.sort()
>>> L
[1, 2, 2, 2, 4, 7, 9]
```

- Une méthode peut être assimilée à une fonction.
- Elle agit sur un objet selon la syntaxe suivante :

`objet.methode(arguments)`

Exemple 2 :

```
>>> phrase = 'Phrase a decouper'
>>> phrase.split('a')
['Phr', 'se ', ' decouper']
>>> phrase.split(' ')
['Phrase', 'a', 'decouper']
>>> mots = phrase.split(' ')
>>> ' '.join(mots)
'Phrase a decouper'
```

- Instruction conditionnelle `if`

if condition :
    bloc d'instructions

TEST SANS ALTERNATIVE

```
1 L = ['toto', 'titi', 'tata']
2
3 if 'toto' in L:
4     L += ['tutu']
5
6 print(L)
```

- Instruction conditionnelle `if`

if condition :
    bloc d'instructions 1
else :
    bloc d'instructions 2

TEST AVEC UNE ALTERNATIVE

```
# a x + b y + c = 0 est l'équation d'une droite
if b == 0:
    print("x =", -c / a)
else:
    pente = -a / b
    ordonnee_origine = -c / b
    print("y =", pente, "x +", ordonnee_origine)
```

- Instruction conditionnelle `if`

if condition 1 :
    bloc d'instructions 1
elif condition 2 :
    bloc d'instructions 2
    ...
else :
    bloc d'instructions n

TEST AVEC ALTERNATIVES MULTIPLES

```
1 if x % 3 == 0:
2     x = x + 1
3 elif x % 3 == 1:
4     x = x - 2
5 else:
6     x = 3 * x
```

- Instructions itératives `for` et `while`

for indice **in range**(debut,fin,pas) :
    bloc d'instructions

BOUCLE FOR

```
1 L = []
2
3 for i in range(0,100,4):
4     L.append(i**2)
5
6 print(L)
```

```
1 L = [i**2 for i in range(0,100,4)]
```

- Instructions itératives `for` et `while`

while condition :
    bloc d'instructions

BOUCLE CONDITIONNELLE WHILE

```
1 s, k = 0, 0
2
3 while s < 2017:
4     k += 1
5     s += k
6
7 print(k)
```

Une structure de données est un mode de représentation des données d'un problème. Le choix d'une telle structure dépend de :

1. la place mémoire qu'elle consomme ;
2. des facilités qu'elle offre pour accéder à certaines données.

Parmi les structures de données classiques, on distingue :

- les structures linéaires

  $\longrightarrow$ représentées par des tableaux unidimensionnels

a_0	a_1	a_2	$\cdots$	$\cdots$	a_n

  On rencontre notamment les piles et les files.

Une structure de données est un mode de représentation des données d'un problème. Le choix d'une telle structure dépend de :

1. la place mémoire qu'elle consomme ;
2. des facilités qu'elle offre pour accéder à certaines données.

Parmi les structures de données classiques, on distingue :

- les matrices ou tableaux multidimensionnels

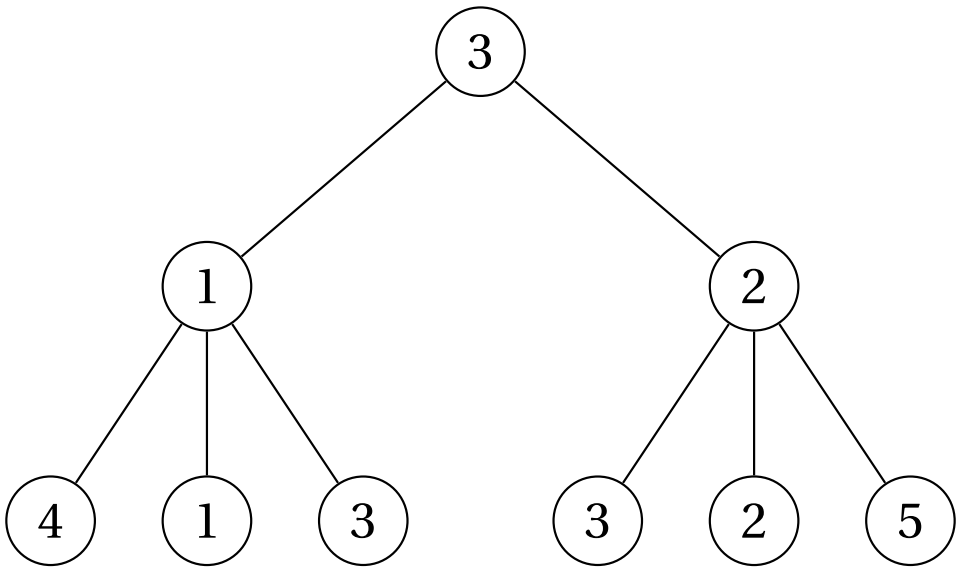
a_{11}	a_{21}	$\cdots$	$\cdots$	a_{n1}
a_{12}	a_{22}	$\cdots$	$\cdots$	a_{n2}
$\vdots$	$\vdots$			$\vdots$
$\vdots$	$\vdots$			$\vdots$
a_{1n}	a_{2n}	$\cdots$	$\cdots$	a_{nn}

Une structure de données est un mode de représentation des données d'un problème. Le choix d'une telle structure dépend de :

1. la place mémoire qu'elle consomme ;
2. des facilités qu'elle offre pour accéder à certaines données.

Parmi les structures de données classiques, on distingue :

- les structures arborescentes

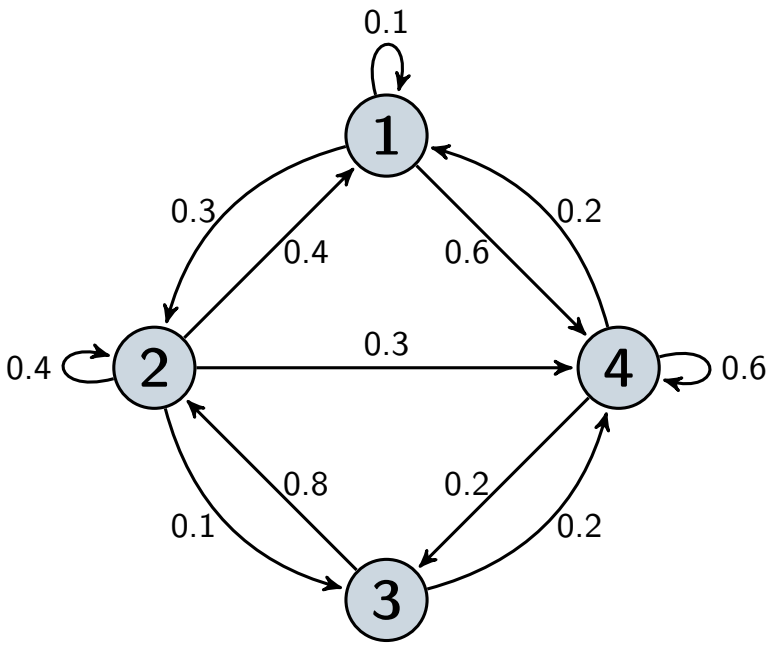


Une structure de données est un mode de représentation des données d'un problème. Le choix d'une telle structure dépend de :

1. la place mémoire qu'elle consomme ;
2. des facilités qu'elle offre pour accéder à certaines données.

Parmi les structures de données classiques, on distingue :

- les structures relationnelles



- Un tableau est une suite finie de valeurs stockées dans des cases mémoires contiguës. Cette structure de données est appelée liste. Ces éléments peuvent être de types distincts.

```
>>> L = ['s', 5, True, 2.1]
>>> type(L)
<class 'list'>
```

La commande `len` permet d'obtenir sa longueur.

```
>>> len(L)
4
```

- On peut construire une liste en concaténant plusieurs listes :

```
>>> ['a', 'b'] + ['c', 'd']
['a', 'b', 'c', 'd']
>>> 4 * ['a', 'b']
['a', 'b', 'a', 'b', 'a', 'b', 'a', 'b']
```

- On peut également construire une liste par compréhension.

```
>>> L = [i**2 for i in range(0,22) if i % 3 == 0]
>>> L
[0, 9, 36, 81, 144, 225, 324, 441]
```

- L[i] permet d'accéder au $(i+1)$-ième élément.
  Principe du slicing (ou tranchage) :

`liste[debut:fin:pas]`

```
>>> L[3] # l'indice initial est 0
81
>>> L[-1] # dernier element de la liste
441
>>> L[1:4] # principe du slicing
[9, 36, 81]
>>> L[len(L):0:-1] # ou bien L[::-1]
[441, 324, 225, 144, 81, 36, 9]
```

- Copie d'une liste

```
>>> L1 = [1, 2, 3, 4, 5, 6]
>>> L2 = L1
>>> L1[2] = 5
>>> L2
[1, 2, 5, 4, 5, 6]
```

ERREUR À NE PAS COMMETTRE !

```
>>> L1 = [1, 2, 3, 4, 5, 6]
>>> L1[:]
[1, 2, 3, 4, 5, 6]
>>> L2 = L1[:]
>>> L1[2] = 5
>>> L2
[1, 2, 3, 4, 5, 6]
```

BONNE MÉTHODE

- Ajout d'un élément, suppression :

```
>>> L = list(range(1,8))
>>> L
[1, 2, 3, 4, 5, 6, 7]
>>> L.append(2)
>>> L
[1, 2, 3, 4, 5, 6, 7, 2]
>>> L.remove(5) # on retire 5 de la liste
>>> L
[1, 2, 3, 4, 6, 7, 2]
>>> L.pop() # on retire le dernier element
2
>>> L
[1, 2, 3, 4, 6, 7]
```

- Parcours des éléments d'une liste :

On peut parcourir directement tous les éléments d'un tableau :

for element **in** L :
bloc d'instructions

```
>>> import random
>>> L = [random.randint(0,9) for i in range(15)]
>>> L
[4, 1, 2, 2, 0, 9, 2, 5, 4, 5, 0, 1, 9, 5, 0]
>>> s = 0
>>> for elem in L:
...     s += elem
...
>>> s
49
```

- Un critère important de choix d'algorithme est la minimisation du temps d'exécution.
- La complexité est une mesure du nombre d'opérations élémentaires.
- On appelle ici opération élémentaire :
  1. un accès en mémoire pour lire ou écrire la valeur d'une variable
  2. une opération arithmétique entre entiers ou flottants
  3. une comparaison entre caractères
  4. une opération booléenne, *etc.*
- Exemples de calcul de complexité :

```
>>> s = 0
>>> for k in range(12):
...     s += k
...
>>> print(s)
66
```

EXEMPLE 1

- Un critère important de choix d'algorithme est la minimisation du temps d'exécution.
- La complexité est une mesure du nombre d'opérations élémentaires.
- On appelle ici opération élémentaire :
  1. un accès en mémoire pour lire ou écrire la valeur d'une variable
  2. une opération arithmétique entre entiers ou flottants
  3. une comparaison entre caractères
  4. une opération booléenne, *etc.*
- Exemples de calcul de complexité :

```
>>> def diviseurs(n):
...     for k in range (1,n+1):
...         if n % k == 0:
...             print(k)
...
```

EXEMPLE 2

- Un critère important de choix d'algorithme est la minimisation du temps d'exécution.
- La complexité est une mesure du nombre d'opérations élémentaires.
- On appelle ici opération élémentaire :
  1. un accès en mémoire pour lire ou écrire la valeur d'une variable
  2. une opération arithmétique entre entiers ou flottants
  3. une comparaison entre caractères
  4. une opération booléenne, *etc.*
- Exemples de calcul de complexité :

```
>>> L = [(a,b) for a,b in [(i,j) for i in range(5)
... for j in range(5) if i>j] if a+b > 3]
```

EXEMPLE 3

- On cherche seulement à calculer un ordre de grandeur du nombre d'opérations élémentaires :
  1. les différentes opérations élémentaires ne demandent pas toutes le même temps de calcul ;
  2. le même programme peut être exécuté sur deux machines différentes.

  Obtenir une valeur exacte n'aurait donc pas grand sens.

- On cherche seulement à calculer un ordre de grandeur du nombre d'opérations élémentaires :
  1. les différentes opérations élémentaires ne demandent pas toutes le même temps de calcul ;
  2. le même programme peut être exécuté sur deux machines différentes.

  Obtenir une valeur exacte n'aurait donc pas grand sens.
- Bien faire la distinction entre $f(n) = \mathrm{O}(g(n))$ et $f(n) = \vartheta(g(n))$.
- Il a plusieurs types de complexité :
  1. la complexité dans le pire des cas : il s'agit d'un majorant de la complexité, obtenue dans le cas le plus coûteux en opérations. C'est souvent celle qui est considérée.
  2. la complexité dans le meilleur des cas : on calcule au contraire le nombre d'opérations dans le cas le plus favorable.
  3. la complexité en moyenne : on compte en moyenne le nombre d'opérations élémentaires effectuées. Ce calcul est souvent difficile !

n	10	20	30	40
$\log(n)$	0,001 s	0,0013 s	0,0015 s	0,0016 s
n	0,01 s	0,02 s	0,03 s	0,04 s
n^2	0,1 s	0,4 s	0,9 s	1,6 s
n^5	1,7 mn	53,3 mn	6,75 h	28,3 h
2^n	1 s	17,5 mn	12,4 jours	34,9 ans
10^n	116 jours	3×10^7 siècles	3×10^{17} siècles	3×10^{27} siècles
$n!$	1 h	$7,7 \times 10^5$ siècles	$8,4 \times 10^{19}$ siècles	$2,6 \times 10^{35}$ siècles
n^n	116 jours	$3,3 \times 10^{13}$ siècles	$6,5 \times 10^{31}$ siècles	$3,8 \times 10^{51}$ siècles

TEMPS DE CALCUL POUR UN ORDINATEUR EFFECTUANT 1000 OPÉRATIONS ÉLÉMENTAIRES PAR SECONDE