COURS 9 - Les Algorithmes - Partie 1

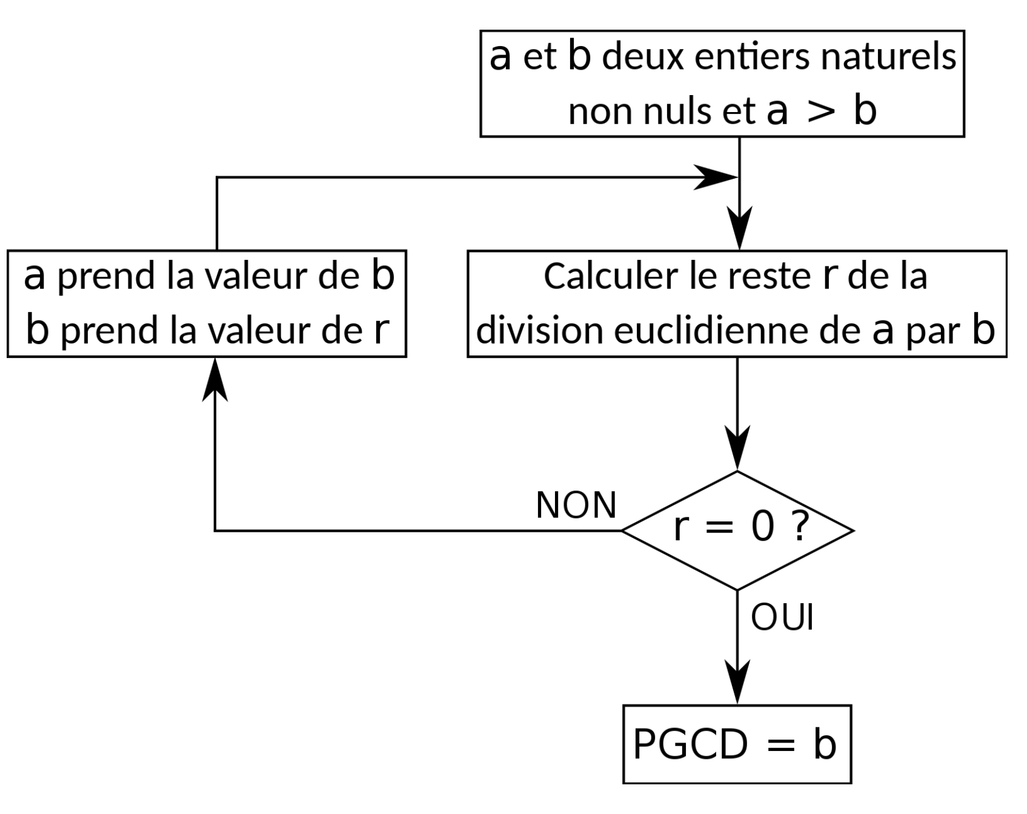
1. Introduction

Abu Abdallah Muhammad Ibn Musa Al-Khwârizmî (780-850), le "père de l'algèbre" était un savant de Bagdad, originaire du Khwârizm, une région d'Asie Centrale, actuel Ouzbekistan. Ses écrits en langue arabe ont permis la diffusion jusqu'en Europe des chiffres arabes et de l'algèbre, mot qui a pour origine le titre d'un de ses ouvrages. Ses écrits seront traduits en latin vers le XIIe siècle. Il a classifié les algorithmes existant à son époque et son nom est à l'origine du mot algorithme. Dans l'un de ses ouvrages, il présente les équations canoniques de degré inférieur ou égal à 2, avec des exemples. Puis il propose des algorithmes de résolution pour ces équations.

L'algorithme d'Euclide est peut-être le plus ancien algorithme non trivial. On le trouve dans le livre VII des Éléments, premier traité écrit de mathématiques, datant d'environ 2300 ans. Mais des algorithmes étaient déjà utilisés dans le calcul à Babylone, au sud de Bagdad dans l'actuel Irak. Donald Knuth a eu l'occasion d'étudier des tablettes datant d'environ 2000 ans avant notre ère, avec des calculs effectués en base 60 et des nombres écrits en virgule flottante !

L'algorithme d'Euclide est utilisé pour le calcul du pgcd de deux entiers *m* et *n*

- **Étape 1** : on divise m par n et on note r le reste (0 ≤ r < n).
- **Étape 2** : si r = 0, c'est terminé, le pgcd est n.
- **Étape 3** : sinon, on remplace m et n par n et r et on recommence à l'étape 1.



Le programme en Python :

In [1]:
```
def pgcd(m, n):
    r = m % n
    while r != 0:
        m, n = n, r
        r = m % n
    return n


pgcd(2344,5676)
```

Out[1]:
```
4
```

Donald Knuth énonce quelques règles dans un ouvrage monumental : *The Art of Computer Programming* dont l'écriture a commencé en 1962 et la publication du premier volume date de 1968. L'ouvrage commence par un algorithme décrivant la manière de lire le premier volume ! de cet ensemble de livres puis de lire les différents volumes !

Après quelques pages, il présente cinq caractéristiques importantes d'un algorithme :

- Un algorithme doit toujours se terminer après un nombre fini d'étapes.
- Chaque étape d'un algorithme doit être définie précisément, les actions à mener doivent être spécifiées rigoureusement et sans ambiguïté pour chaque cas.
- Un algorithme a des entrées, zéro ou plus, quantités qui lui sont données avant ou pendant son exécution.
- Un algorithme a une ou plusieurs sorties, quantités qui ont une relation spécifiée avec les entrées.
- Les instructions doivent être suffisamment basiques pour pouvoir être en principe exécutées de manière exacte et en un temps fini par une personne utilisant un papier et un crayon.

Une question à se poser quand on veut écrire un algorithme :

Si on disposait de tout le temps nécessaire, comment ferait-on avec un papier et un crayon, en écrivant tous les détails de notre action jusqu'à l'objectif final, de telle sorte qu'une autre personne puisse les reproduire de manière exactement similaire ?

2. Les outils

2.1. Compteurs et accumulateurs

Un compteur, comme son nom l'indique, sert à compter. Par exemple, on peut compter le nombre d'essais dans un jeu. De manière générale, c'est une variable initialisée à 0 qui est incrémentée d'une unité à chaque passage dans une boucle, éventuellement suite à un test. Nous allons voir quelques exemples.

Compteur et boucle conditionnelle

- Sans test :

In [2]:
```
def taille(n):
    cpt = 0
    while n > 0:
        cpt = cpt + 1
        n = n // 2
    return cpt


taille(1000)
```

Out[2]:
```
10
```

On compte le nombre de divisions euclidiennes successives de n par 2, jusqu'à arriver à un quotient nul. On obtient donc le nombre de chiffres dans l'écriture binaire de n.

- Avec test :

In [3]:
```
def nombre_de_1(n):
    cpt = 0
    while n > 0:
        if n % 2 == 1:
            cpt = cpt + 1
        n = n // 2
    return cpt


nombre_de_1(1000)
```

Out[3]:
```
6
```

Le programme est identique au précédent, mais on incrémente le compteur seulement quand un reste vaut 1 (`if n % 2 == 1`). On compte donc le nombre de 1 dans l'écriture binaire de n.

Compteur et boucle inconditionnelle

Dans une boucle inconditionnelle sans test, un compteur compterait le nombre de passages dans la boucle. Or, ce nombre est connu à l'avance et donc un compteur n'apporterait rien.

Voyons donc avec test :

In [4]:
```
def diviseurs(n):
    cpt = 0
    for d in range(1, n + 1):
        if n % d == 0:
            cpt = cpt + 1
    return cpt


diviseurs(1000)
```

Out[4]:
```
16
```

Le compteur est incrémenté quand *n* est divisible par *d*. Il compte donc le nombre de diviseurs de n qui est le résultat renvoyé par la fonction.

Accumulateurs

Un accumulateur est semblable à un compteur mais il est en général incrémenté d'une valeur différente de 1. Il peut aussi être décrémenté, par exemple dans le calcul d'une somme de termes.

Supposons maintenant la fonction suivante dans laquelle liste est une liste de nombres.

- Sans test :

In [5]:
```
def somme(liste):
    acc = 0
    for x in liste:
        acc = acc + x
    return acc


somme([4,6,7,100])
```

Out[5]:
```
117
```

La fonction renvoie la somme des nombres contenus dans la liste.

- Avec test :

In [6]:
```
def somme(liste):
    acc = 0
    for x in liste:
        if x%2==0 :
            acc = acc + x
    return acc


somme([4,6,7,100])
```

Out[6]:
```
110
```

Saisissez votre réponse ici :

La fonction renvoie la somme des nombres pairs contenus dans la liste.

2.2. Permutation de valeurs

On est souvent amené à devoir permuter des valeurs entre des variables, en particulier échanger les valeurs de deux variables. Cet échange est présent dans les algorithmes de tri exposés au chapitre suivant qui reposent sur la comparaison de deux valeurs et en fonction du résultat leur éventuelle permutation. Dans la construction d'une suite de nombres définie par une relation du type $\mathbf{u_{n+2} = f(u_{n+1}; u_n)}$, on calcule un terme en fonction des deux précédents. Ceci est possible en utilisant seulement deux variables puisque cela revient à passer du couple $\mathbf{(u_{n+1}; u_n)}$ au couple $\mathbf{(u_{n+2}; u_{n+1})}$.

Avant tout, que penser du code suivant ?

In [7]:
```
var1 = 17
var2 = 23
var1 = var2
var2 = var1


var1,var2
```

Out[7]:
```
(23, 23)
```

Saisissez votre réponse ici :

La troisième affectation, var1 prend la valeur de var2 soit 23

La quatrième affectation, var2 prend la valeur var1 soit 23 (puisque var 1 vient d'être modifié)
</mark>

Conclusion :
Avec ce code, les deux variables prendront la même valeur, la valeur initiale de var2.

On comprend qu'il faut modifier la valeur d'une variable sans perdre sa valeur initiale qu'il faut donc stocker dans une troisième variable.

In [8]:
```
var1 = 17
var2 = 23
temp=var1
var1 = var2
var2 = temp


var1,var2
```

Out[8]:
```
(23, 17)
```

La valeur de var1, 17 est gardée dans la nouvelle variable temp.
Ce nom est choisi car cette variable est temporaire, elle n'est utilisée que le temps de l'échange.

On peut alors modifier la valeur de var1 en lui affectant la valeur de var2, et finalement affecter à var2 la valeur de temp (valeur initiale de var1 qui n'a pas été perdue.)

Ce procédé est utilisé dans de nombreux langages, certains langages possèdent une fonction *swap* pour permuter deux valeurs.

En Python, c'est l'existence du type **tuple** qui nous permet d'avoir une instruction simplifiée :

In [9]:
```
var1 = 17
var2 = 23
var1, var2 = var2, var1


var1,var2
```

Out[9]:
```
(23, 17)
```

Expliquez le comportement de ce code ici :

Les valeurs de var1 et var2 ont été permutées sans avoir à passer par une troisième variable temporaire.

2.3. Tests et boucles

Les tests

Dans la plupart de nos programmes, nous trouvons des tests qui utilisent les structures `if ...` , `if ... else ...` ou `if ... elif ... else ...`

Première remarque : else n'est jamais suivi d'une expression. Une expression après else signifierait "sinon, si l'expression a la valeur True" et nous sommes alors dans le cadre d'une structure elif. "Sinon" signifie : si ce qui est avant est faux".

Par exemple :

In [10]:
```
x=-2

if x > 0:
    x = x - 3
elif x < 0:
    x = x + 5
else:
    x = x + 2


x
```

Out[10]:
```
3
```

Nous avons trois cas distincts : le premier cas est x > 0, le deuxième cas est x < 0, le troisième cas regroupe tout ce qui n'est ni dans le premier cas ni dans le deuxième, donc si x est nul.

Donc :

- Si la valeur initiale de x est -2, elle est actualisée à 3
- Si la valeur initiale de x est 2, alors elle est actualisée à -1
- Si la valeur initiale de x est 0, elle est actualisée à 2

Une deuxième remarque : la structure `if ... if ... else ...` n'est pas équivalente à la structure `if ... elif ... else ...` En parlant, nous pouvons énoncer : si x est strictement positif, ..., puis si x est strictement négatif, ..., sinon, ... Et certains comprendront peut être que sinon correspond x nul.
Ce n'est évidemment pas le cas.

Examinons le code qui suit :

In [11]:
```
if x > 0:
    x = x - 3
if x < 0:
    x = x + 5
else:
    x = x + 2
```

Quelle est la valeur finale de x si la valeur initiale est -2 ? Pourquoi ?

Saisissez votre réponse ici :

Puisque la valeur de x est strictement négative, elle est actualisée à 3 (-2 + 5).

Quelle est la valeur finale de x si la valeur initiale est 2 ? Pourquoi ?

Saisissez votre réponse ici :

Puisque la valeur de x est strictement positive, elle est actualisée à -1 (2 - 3). Ensuite, puisque la valeur courante de x est strictement négative, elle est actualisée à 4 (-1 + 5).

Troisième remarque : L'expression qui suit `if` a une valeur **True** ou **False**, ou une valeur qui peut être interprétée comme True ou False.

Prenons par exemple `if x > 0`. Dans cet exemple, nous n'écrirons pas `if (x > 0) == True`.
Donc il en est de même si l'expression est composée d'une seule variable booléenne ou d'un appel de fonction renvoyant un booléen.
Nous écrirons `if b` et `if f(x)` et non pas `if b == True` ou `if f(x) == True`. Ces deux dernières écritures reviendraient à tester `True == True` ou `False == True`.

Les boucles

Nos programmes sont constitués de boucles et même de boucles imbriquées (elles sont utilisées particulièrement avec des listes de listes ou dans les algorithmes de tri). Nous pouvons avoir une ou plusieurs boucles while ou for à l'intérieur d'une boucle while ou for.

Par exemple :

In [12]:
```
for i in range(4):
    for j in range(3):
        print(i + j)
```

```
0
1
2
1
2
3
2
3
4
3
4
5
```

Les valeurs successives des variables i et j sont :

i = 0 et j = 0, puis j = 1, puis j = 2, ensuite
i = 1 et j = 0, puis j = 1, puis j = 2, ensuite
i = 2 et j = 0, puis j = 1, puis j = 2, ensuite
i = 3 et j = 0, puis j = 1, puis j = 2.

Pour chacune des quatre valeurs de i (0, 1, 2, 3), j prend trois valeurs, (0, 1, 2) et nous aurons donc douze affichages avec la fonction print.

2.4. Exercices

Pour chacun de ces 3 scripts, déterminez en fonction de n le nombre d'additions effectuées.

In [13]:
```
x=0
for i in range(2) :
    x=x+i
    for j in range (3):
        x=x+j
```

Saisissez votre réponse ici :

La première boucle for va être exécutée 2 fois (range(2)) : 2 additions
Le seconde boucle for va être exécutée 3 fois (range(3)) dans la première boucle (range(2)) : 6 additions
Le total est donc de 8 additions

In [ ]:
```
x=0
for i in range(2) :
    x=x+i
    for j in range (n):
        x=x+j
```

Saisissez votre réponse ici :

On fait le même type de calcul que pour l'exercice 1 :
La première boucle for va être exécutée 2 fois (range(2)) : 2 additions
Le seconde boucle for va être exécutée n fois (range(n)) dans la première boucle (range(2)) : 2n additions
Le total est donc de 2n + 2 additions

In [ ]:
```
x=0
for i in range(n) :
    x=x+i
    for j in range (n):
        x=x+j
```

Saisissez votre réponse ici :

Cette fois :
La première boucle for va être exécutée n fois (range(n)) : n additions
Le seconde boucle for va être exécutée n fois (range(n)) dans la première boucle (range(n)) : n x n additions
Le total est donc de n + n^2 additions</amrk>