NSI en première (2019-2020)
Algorithmes 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 XII^{e} 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 !

Algorithme d'Euclide
Il 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 ($0 \leq r < n$).
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.

Programme en Python :

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

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 :

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

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 :

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

Le programme est identique au précédent, mais on incrémente le compteur seulement quand un reste vaut 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 une boucle inconditionnelle avec un test.

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

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.

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.

Nous supposons que `liste` est une liste de nombres.

Sans test :

```
def somme(liste):
    acc = 0
    for x in liste:
        acc = acc + x
    return acc
```

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

Avec test :

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

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 $u_{n+2} = f(u_{n+1}, u_n)$, on calcule un terme en fonction des deux précédents. Ceci est possible en utilisant seulement deux variables puisque cela revient à passer du couple (u_{n+1}, u_n) au couple (u_{n+2}, u_{n+1}).

Le principe général est simple. Avant tout, que penser du code suivant ?

```
var1 = 17
var2 = 23
var1 = var2
var2 = var1
```

À la troisième affectation, `var1` prend la valeur courante de `var2` donc 23. La quatrième affectation utilise la valeur courante de `var1`, qui est 23 à ce moment, donc `var2` prend la valeur 23. Les deux variables ont finalement pour valeur 23, soit la valeur initiale de `var2`.

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

```
var1 = 17
var2 = 23
temp = var1
var1 = var2
var2 = temp
```

La valeur de `var1`, 17 est gardée dans `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 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 : `var1, var2 = var2, var1`. Cette instruction signifie que le couple `(var1, var2)` prend la valeur du couple `(var2, var1)` c'est-à-dire la valeur `(23, 17)`. Donc `var1` prend la valeur 23 et `var2` prend la valeur 17. L'échange des valeurs est réalisé.

2.3 Tests et boucles

Les tests

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

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

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. L'écriture `else x == 0` provoquerait une erreur.

Par exemple : si la valeur initiale de `x` est 5, alors elle est actualisée à 2. Si la valeur initiale de `x` est -2, elle est actualisée à 3. 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 à `x` nul. Ce n'est évidemment pas le cas.

Examinons le code qui suit :

```
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 5 ? Nous avons ici deux blocs distincts : le bloc `if x > 0 ...` et le bloc `if x > 0 ... else ...`. Donc puisque la valeur de `x` est strictement positive, elle est actualisée à 2. Ensuite, puisque la valeur courante de `x` est strictement positive, elle est actualisée à 4.

Si la valeur initiale de `x` est 2, elle est strictement positive donc elle est actualisée à -1. Puis elle est strictement négative donc elle est actualisée à 4.

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 au chapitre 8). Nous pouvons avoir une ou plusieurs boucles `while` ou `for` à l'intérieur d'une boucle `while` ou `for`.

Par exemple :

```
for i in range(4):
    for j in range(3):
        print(i + j)
```

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`.

3 Validité et coût

Lorsqu'on écrit un algorithme, il est impératif de vérifier que cet algorithme va produire un résultat en un temps fini et que ce résultat sera correct dans le sens où il sera conforme à une spécification précise. Nous dirons alors que l'algorithme est valide.

3.1 Validité d'un algorithme itératif

Correction

Un algorithme itératif est construit avec des boucles. Pour prouver qu'il est correct, nous disposons de la notion d'invariant de boucle.

Définition

Un invariant d'une boucle est une propriété qui est vérifiée avant l'entrée dans une boucle, à chaque passage dans cette boucle et à la sortie de cette boucle. On peut faire le lien avec les suites définies par récurrence du programme de mathématiques.

Pour démontrer qu'une propriété est un invariant d'une boucle, on utilise un raisonnement semblable au raisonnement par récurrence.

On commence par vérifier que la propriété est vraie avant l'entrée dans la boucle. Cette étape s'appelle l'initialisation. On prouve ensuite que si la propriété est vraie avant un passage dans la boucle, alors elle est vraie après ce passage. Cette étape s'appelle l'hérédité. On peut alors conclure, que la propriété est vraie à la sortie de la boucle.

Exemple

Voici un algorithme de calcul avec une boucle conditionnelle et deux variables `a` et `b`, `a` ayant pour valeur un entier naturel :

```
m = 0
p = 0
tant que m < a
    m = m + 1
    p = p + b
fin du tant que
```

Notons m et p les valeurs des variables `m` et `p`.

Nous allons montrer que la propriété "$p = m \times b$" est un invariant de la boucle "tant que".

Avant le premier passage dans la boucle, $m = 0$ et $p = 0$, donc l'égalité $p = m \times b$ est vraie.

Supposons que $p = m \times b$ avant un passage dans la boucle. Les nouvelles valeurs de `m` et `p` après le passage, notées m' et p' vérifient : $m' = m+1$ et $p' = p+b$. Alors $p' = m \times b + b = (m+1) \times b = m' \times b$. Donc la propriété est vraie après ce passage dans la boucle.

Nous pouvons conclure qu'à la sortie de la boucle, $p = m \times b$. Et puisqu'à la sortie de la boucle, la variable `m` a pour valeur celle de `a`, nous avons finalement obtenu le produit $p = a \times b$.

Terminaison

Un algorithme ne doit toujours comporter qu'un nombre fini d'étapes. Afin de prouver la terminaison d'un algorithme itératif, (qui contient une boucle), nous utilisons la notion de variant. Nous parlons ici de boucles conditionnelles. Dans le cas de boucles non conditionnelles, le nombre d'étapes est déterminé.

Méthode

On choisit un variant, c'est-à-dire une expression, la plus simple étant une variable, telle que la suite formée par les valeurs de cette expression au cours des itérations converge en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt.

Considérons par exemple le code suivant où la valeur de la variable `a` est un nombre quelconque :

```
x = 0
while x ** 2 < a:
    x = x + 1
```

Si la valeur de `a` est négative ou nulle, il n'y a aucun passage dans la boucle. Sinon, la suite des valeurs de la variable `x`, le variant choisi, est 0, 1, 2, ..., n, et n est certainement la première valeur supérieure ou égale à la racine carrée de la valeur de `a`. Le nombre de passages dans la boucle est donc fini.

Revenons sur l'exemple du produit de deux nombres étudié plus haut. Nous avons prouvé qu'en sortie de boucle, la valeur de `p` était le produit $a \times b$. Mais nous n'avons pas prouvé la terminaison, c'est-à-dire que la sortie de boucle était effective après un nombre fini de passages. Pour cela, dans cet exemple, nous choisissons comme variant la variable `m`. Cette variable prend pour valeurs successives 0, 1, ..., a et il y a donc exactement a passages dans la boucle, ce qui prouve la terminaison.

3.2 Coût

Un programme doit traiter une liste de 10^7 éléments puis une liste de 10^8 éléments. Est-ce que le temps d'exécution du programme sera multiplié par 10 ? Quel est le rapport entre le temps d'exécution est la taille de la liste ? Ce sont des questions auxquelles il faut réfléchir quand on écrit un algorithme.

Les réponses sont variées et dépendent de l'algorithme et de la liste. Pour une liste donnée, un programme peut être plus rapide qu'un autre, mais avec une autre liste, ce peut être le contraire. Le même programme peut être plus rapide avec la liste la plus longue.

De plus, pour traiter un même problème, non seulement nous pouvons disposer de plusieurs algorithmes mais un même algorithme peut avoir un temps d'exécution différent selon le langage de programmation utilisé et suivant la machine sur laquelle le programme est exécuté. L'étude n'est pas simple à

réaliser et pour comparer deux algorithmes nous allons ici nous concentrer sur le nombre d'opérations à effectuer en essayant d'évaluer un ordre de grandeur de ce nombre en fonction de la taille des données.

Nous parlerons du coût d'un algorithme ou de sa complexité. Ce coût pouvant être très différent pour une même taille de données, nous nous placerons dans le pire des cas, celui où le coût est le plus important.

Étudions quelques exemples simples qui sont à la base de ce type d'étude. Nous commençons par un algorithme rencontré précédemment.

```
m = 0
p = 0
tant que m < a
    m = m + 1
    p = p + b
fin du tant que
```

Les passages dans la boucle ont lieu pour les valeurs de `m` égales à 0, 1, 2, ..., $a-1$ si la valeur de la variable `a` est un entier naturel a. Nous avons donc exactement a passages dans la boucle. À chaque passage, nous comptons deux additions ainsi que deux affectations. Nous pouvons donc dire que le nombre d'additions est $2a$ ou que le nombre d'opérations, au sens large en comptant les affectations, est $4a$. Nous dirons alors que le coût est proportionnel à a ou qu'il est linéaire.

Avec une boucle "tant que", le calcul peut être plus compliqué puisque le nombre de passages dans la boucle varie avec les cas pour une même taille de données. Nous devons alors identifier le pire des cas, c'est-à-dire celui où le nombre de passages est maximal.

Considérons une boucle "pour" où le nombre de passages dans la boucle est bien déterminé.

```
somme = 0
for i in range(1, n+1):
    somme = somme + i
```

Cet algorithme, ou ce programme, permet de calculer la somme des entiers de 1 à `n`. Il y a clairement `n` passages dans la boucle. À chaque passage nous avons une addition et une affectation, donc un total de `n` additions et `n` affectations. Nous pouvons affirmer que le coût est linéaire.

Complexité linéaire

Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $\alpha n + \beta$, avec α et β réels, $\alpha > 0$, nous disons que l'algorithme a un coût linéaire ou une complexité linéaire.

Dans le cas de deux boucles "for" imbriquées, nous avons trois cas typiques. Dans un cas, le coût est linéaire. Dans les deux autres cas, nous disons que le coût est quadratique.

Complexité quadratique

Soit n la taille d'une donnée. Si le nombre d'opérations à effectuer peut s'écrire $\alpha n^2 + \beta n + \gamma$, avec α, β et γ réels, $\alpha > 0$, nous disons que l'algorithme a un coût quadratique ou une complexité quadratique.

Dans les codes qui suivent, les pointillés sous-entendent un nombre fixe d'opérations.

Premier cas : `n` est la taille de la donnée, `k` est un nombre fixé.

```
for i in range(n):
    ...
    for j in range(k):
        ...
```

Nous avons `n` passages dans la boucle externe. À chaque passage, nous avons un nombre fixe d'opérations q puis `k` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de `i`, nous avons $q + k \times r = \alpha$ opérations. Le nombre total d'opérations est donc αn et le coût est linéaire.

Deuxième cas : `n` est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(n):
        ...
```

Nous avons `n` passages dans la boucle externe. À chaque passage, nous avons un nombre fixe d'opérations q puis `n` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de `i`, nous avons $q + n \times r$ opérations. Le nombre total d'opérations est donc $n(q + n \times r) = rn^2 + qn$ et le coût est quadratique.

Troisième cas : `n` est la taille de la donnée.

```
for i in range(n):
    ...
    for j in range(i):
        ...
```

Nous avons `n` passages dans la boucle externe. À chaque passage, pour chaque valeur de `i`, nous avons un nombre fixe d'opérations q puis `i` passages dans la boucle interne. Dans la boucle interne, nous avons un nombre fixe d'opérations r. Donc pour chaque valeur de `i`, nous avons $q + i \times r$ opérations. Les valeurs de `i` sont successivement 0, 1, 2, ..., n -1. Le nombre total d'opérations est donc $q + (q + 1 \times r) + (q + 2 \times r) + \ldots + (q + (n-1) \times r)$, soit $nq + r(1 + 2 + \ldots + (n-1))$. Le calcul de la somme des entiers est connu. Le résultat est $nq + r\dfrac{n(n-1)}{2}$. Finalement le nombre d'opérations est $\dfrac{r}{2}n^2 + (q - \dfrac{r}{2})n$ de la forme $\alpha n^2 + \beta n$ et le coût est quadratique.