3 Algorithmique élémentaire

Dans ce chapitre, nous introduisons les concepts élémentaires de l'algorithmique. Nous voyons tout d'abord les structures élémentaires à partir desquelles sont construits tous les algorithmes écrits dans un langage impératif. Ces structures ont déjà été introduites au cours du chapitre 1 dans le cadre de Python. Nous donnons ensuite quelques algorithmes simples intervenant très souvent comme brique d'une construction plus imposante. Ces petits algorithmes sont à bien maîtriser : il faut être capable de les implémenter rapidement lorsque le besoin se fait sentir, éventuellement en les adaptant à la situation.

I Qu'est-ce qu'un algorithme

I.1 Définition

Le mot *Algorithme* vient du nom du mathématicien arabe *Al Khwarizmi*, auteur au IX^{e} siècle d'un ouvrage faisant la synthèse de la résolution des équations du second degré, suivant le signe des coefficients (afin d'éviter l'usage du signe moins). L'ouvrage en question, proposant des méthodes de résolution par manipulations algébriques (réduction à des formes connues) a donné son nom à l'*algèbre*. Les méthodes exposées peuvent s'apparenter à des algorithmes : on y expose, par disjonction de cas (structure conditionnelle) des façons systématiques de résoudre un certain problème, ne laissant ainsi rien au hasard. Il s'agit bien d'un algorithme de résolution.

Définition 3.1.1 (Algorithme)

Un algorithme est une succession d'instructions élémentaires, faciles à faire et non ambiguës, déterminées de façon unique par les données initiales, et fournissant la réponse à un problème posé.

Le développement de l'informatique a marqué l'essor de l'algorithmique, mais cette discipline n'est pas l'apanage de l'informatique. La notion d'algorithme est liée mathématiquement à la possibilité de résolution systématique d'un problème, donc à la notion de méthode de calcul. On trouve dans le domaine purement mathématique de nombreux algorithmes :

- tous les algorithmes de calcul des opérations élémentaires (addition posée, multiplication posée...)
- l'algorithme de la division euclidenne par différences successives
- l'algorithme d'Euclide du calcul du pgcd
- l'algorithme de résolution des équations de degré 2
- l'algorithme du pivot de Gauss pour résoudre les systèmes d'équations linéaires, et répondre à d'autres questions d'algèbre linéaire.
- l'algorithme de Hörner pour l'évaluation d'un polynôme
- etc.

Les questions qu'on peut se poser sont alors les suivantes :

1. Quelles sont les structures élémentaires à partir desquelles sont construits les algorithmes.
2. L'algorithme s'arrête-t-il ? (problème de la terminaison)
3. L'algorithme renvoie-t-il le résultat attendu ? (problème de la correction)
4. Combien de temps dure l'exécution de l'algorithme, notamment lorsqu'on le lance sur de grandes données ? (problème de la complexité).

Nous nous intéressons dans ce chapitre à la première question, les 3 autres faisant l'objet d'un chapitre ultérieur (analyse des algorithmes)

I.2 Le langage

L'étude algorithmique formelle nécessite de se dégager de toute contrainte idiomatique relevant des spécificités de tel ou tel langage. Pour cela, nous utiliserons un pseudo-code, indépendant de toute implémentation. Les traductions ultérieures dans un langage de programmation spécifique se font alors sans difficulté, sachant que certains langages offrent parfois certaines possibilités supplémentaires (qui sont essentiellement du confort, mais n'ajoutent rien à la description formelle des algorithmes, comme par exemple le fait de pouvoir construire une boucle `for` sur un objet itérable queconque en Python).

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales (variables d'entrée, préciser leur type) ;
3. définissant la sortie (variables de résultat, préciser leur type) ;
4. donnant le bloc d'instructions définissant l'algorithme.

La structure générale est donc la suivante :

Algorithme 3.1 : Nom ou description de l'algorithme

Entrée : *a*,*b*,... : type
Sortie : *c*,*d*,... : type

instructions ;
renvoyer (*c*, *d*,...)

La dernière ligne, permet de définir le ou les résultats.

I.3 Les structures élémentaires

Un algorithme repose sur certains types d'instructions élémentaires. Une instruction est une phrase du langage de programmation indiquant à l'ordinateur une ou plusieurs actions à effectuer, induisant un changement d'état de l'ordinateur (c'est-à-dire une modification de la mémoire).
Les différentes structures élémentaires à partir desquelles sont contruits les algorithmes en programmation impérative sont, en plus de l'évaluation d'expressions, de l'utilisation de fonctions antérieures, et de l'affectation (notée $x \leftarrow$ valeur) :

1. **La séquence :**

   Il s'agit d'un regroupement d'instructions élémentaires, qui seront exécutées successivement. Il s'agit de la notion de *bloc* en informatique, délimité suivant les langages par des balises de début et fin (`begin`, `end`), ou tout simplement par l'indentation, comme en Python.

   L'intérêt de la séquence est de pouvoir considérer plusieurs instructions comme une seule, et de pouvoir inclure cette succession dans des structures plus complexes (structures conditionnelles, répétitives...)

   Nous écrirons un bloc de la façon décrite dans l'algorithme 3.2.

   La plupart du temps, un bloc permettra de délimiter la portée d'une structure composée. Dans ce cas, le mot `bloc` sera remplacé par le mot clé de la structure correspondante, par exemple `début pour` et `fin pour`.

Algorithme 3.2 : Bloc

début bloc
| instructions
fin bloc

2. **La structure conditionnelle simple :**

   C'est la structure permettant d'effectuer des disjonctions de cas. Elle permet de distinguer plusieurs cas si ces cas nécessitent des modes opératoires différents. Cela permet également de traiter les cas particuliers.

   La structure basique est un branchement à deux issues :

   `si` *condition* `alors` *instructions* `sinon` *instructions*

   La condition peut être n'importe quel booléen, par exemple le résultat d'un test, ou le contenu d'une variable de type booléen. De plus, la clause alternative (`else`) est le plus souvent facultative (algorithmes 3.3 et 3.4).

Algorithme 3.3 : Structure conditionnelle simple sans clause alternative

Entrée : classe : entier
Sortie : ∅

si *classe = 4* **alors**
| `Afficher('Bestial!')`
fin si

Algorithme 3.4 : Structure conditionnelle simple avec clause alternative

Entrée : classe : entier
Sortie : ∅

si *classe = MPSI4* **alors**
| `Afficher('Bestial!')`
sinon
| `Afficher('Khrass')`
fin si

Certains langages autorisent le branchement multiple, soit *via* une autre instruction (par exemple `case of` en Pascal), soit comme surcouche de l'instruction basique. Nous utiliserons cette possibilité en pseudo-code (algorithme 3.5).

Algorithme 3.5 : Structure conditionnelle multiple

Entrée : classe : entier
Sortie : ∅

si *classe = MPSI4* **alors**
| `Afficher('Bestial!')`
sinon si *classe = PCSI2* **alors**
| `Afficher('Skiii!')`
sinon
| `Afficher('Khrass')`
fin si

Évidemment, d'un point de vue de la complétion du langage, cette possibilité n'apporte rien de neuf, puisqu'elle est équivalente à un emboîtement de structures conditionnelles (algorithme 3.6).

Algorithme 3.6 : Version équivalente

Entrée : classe : entier
Sortie : ∅

si *classe = MPSI4* **alors**
| `Afficher('Bestial!')`
sinon
| **si** *classe = PCSI2* **alors**
| | `Afficher('Skiii!')`
| **sinon**
| | `Afficher('Khrass')`
| **fin si**
fin si

3. **Les boucles**

Une boucle est une succession d'instructions, répétée un certain nombre de fois. Le nombre de passages dans la boucle peut être déterminé à l'avance, ou peut dépendre d'une condition vérifiée en cours d'exécution. Cela permet de distinguer plusieurs types de boucles :

(a) **Boucles conditionnelles, avec condition de continuation.**

On passe dans la boucle tant qu'une certaine condition est réalisée. Le test de la condition est réalisé avant le passage dans la boucle. On utilise pour cela une boucle `while` ou `tant que` en français. L'algorithme 3.7 en est un exemple.

Algorithme 3.7 : Que fait cet algorithme ?

Entrée : ε (marge d'erreur) : réel
Sortie : ∅

$u \leftarrow 1$;
tant que $u > \varepsilon$ **faire**
| $u \leftarrow \sin(u)$
fin tant que

Ici, on calcule les termes d'une suite définie par la récurrence $u_{n+1} = \sin(u_n)$. On peut montrer facilement que cette suite tend vers 0. On répète l'itération de la suite (donc le calcul des termes successifs) tant que les valeurs restent supérieures à ε.

Comme dans l'exemple ci-dessus, une boucle `while` s'utilise le plus souvent lorsqu'on ne connaît pas à l'avance le nombre de passages dans la boucle. Souvent d'ailleurs, c'est le nombre de passages dans la boucle qui nous intéresse (afin, dans l'exemple ci-dessus, d'estimer la vitesse de convergence de la suite). Dans ce cas, il faut rajouter un compteur de passages dans la boucle, c'est-à-dire une variable qui s'incrémente à chaque passage dans la boucle. C'est ce que nous avons fait dans l'algorithme 3.8.

La valeur finale de i nous dit maintenant jusqu'à quel rang de la suite il faut aller pour obtenir la première valeur inférieure à ε (et par décroissance, facile à montrer, toutes les suivantes vérifieront la même inégalité).

(b) **Boucles conditionnelles, avec condition d'arrêt**

Il s'agit essentiellement de la même chose, mais exprimé de façon légèrement différente. Ici, on répète la série d'instructions jusqu'à la réalisation d'une certaine condition. Il s'agit de la boucle `repeat... until...`, ou `répéter... jusqu'à ce que...`, en français. L'algorithme précédent peut se réécrire de la façon suivante, de façon quasi-équivalente, à l'aide d'une boucle `repeat... until...` (algorithme 3.9).

La différence essentielle avec une boucle `while` est que, contrairement à une boucle `while`,

Algorithme 3.8 : Calcul de i tel que $u_i \leqslant \varepsilon$

Entrée : ε (marge d'erreur) : réel
Sortie : i : entier

$u \leftarrow 1$;
$i \leftarrow 0$;
tant que $u > \varepsilon$ **faire**
    $u \leftarrow \sin(u)$;
    $i \leftarrow i + 1$
fin tant que
renvoyer i

Algorithme 3.9 : Vitesse de convergence de u_n

Entrée : ε (marge d'erreur) : réel
Sortie : i (rang tel que $u_i \leqslant \varepsilon$) : entier

$u \leftarrow 1$;
$i \leftarrow 0$;
répéter
    $u \leftarrow \sin(u)$;
    $i \leftarrow i + 1$
jusqu'à ce que $u \leqslant \varepsilon$;
renvoyer i;

on passe nécessairement au moins une fois dans la boucle. À part ce détail, on a équivalence entre les deux structures, en remplaçant la condition de continuation par une condition d'arrêt (par négation). Ainsi, une boucle `repeat instructions until condition` est équivalente à la structure donnée en algorithme 3.10.

Algorithme 3.10 : Structure équivalente à `repeat... until...` : version 1

instructions ;
tant que $\neg$ *condition* **faire**
    instructions
fin tant que

Remarquez ici le passage forcé une première fois dans la succession d'instructions (bloc isolé avant la structure).

On peut éviter la répétition de la succession d'instructions en forçant le premier passage à l'aide d'une variable booléenne (algorithme 3.11)

Algorithme 3.11 : Structure équivalente à `repeat... until...` : version 2

$b \leftarrow$ True ;
tant que *(*$\neg$ *condition)* $\vee$ b **faire**
    instructions ;
    $b \leftarrow$ False
fin tant que

Cela a cependant l'inconvénient d'augmenter le nombre d'affectations.

Réciproquement, une boucle `while condition do instructions` est équivalente à la structure de l'algorithme 3.12.

Au vu de ces équivalences, même si en Python, les boucles `repeat` n'existent pas, nous nous

Algorithme 3.12 : Structure équivalente à `while... do...`

si *condition* **alors**
  répéter
    instructions
  jusqu'à ce que ¬ *condition*;
fin si

autoriseront à décrire les algorithmes en utilisant cette structure, plus naturelle dans certaines situations. Il faut cependant garder à l'esprit que dans le cadre d'une définition formelle d'un algorithme, cela crée une redondance avec la structure `while`.

(c) **Boucles inconditionnelles**

Il s'agit de boucles dont l'arrêt ne va pas dépendre d'une condition d'arrêt testée à chaque itération. Dans cette structure, on connaît par avance le nombre de passages dans la boucle. Ainsi, on compte le nombre de passages, et on s'arrête au bout du nombre souhaité de passages. Le compteur est donné par une variable incrémentée automatiquement, comme dans l'algorithme

Algorithme 3.13 : Cri de ralliement

Entrée : ∅
Sortie : cri : chaîne de caractères

cri ← 'besti' ;
pour $i \leftarrow 1$ **à** 42 **faire**
  cri ← cri + 'â'
fin pour
cri ← cri + 'l' ;
renvoyer *cri*

I.4 Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme. L'intérêt est multiple :

- Éviter d'avoir à écrire plusieurs fois la même séquence d'instructions, si elle est utilisée à différents endroits d'un algorithme. On peut même utiliser une même procédure dans différents algorithmes principaux.
- Une procédure peut dépendre de paramètres. Cela permet d'adapter une séquence donnée à des situations similaires sans être totalement semblables, les différences entre les situations étant traduites par des valeurs différentes de certains paramètres.
- Écrire des procédures permet de sortir la partie purement technique de l'algorithme principal, de sorte à dégager la structure algorithmique de ce dernier de tout encombrement. Cela augmente la lisibilité de l'algorithme.
- Une procédure peut s'appeler elle-même avec des valeurs différentes des paramètres (récursivité). Cela permet de traduire au plus près certaines définitions mathématiques par récurrence. Cela a un côté pratique, mais assez dangereux du point de vue de la complexité si on n'est pas conscient précisément de ce qu'implique ce qu'on écrit.

La procédure 3.14 est un exemple typique de procédure : il s'agit de l'affichage d'un polynôme entré sous forme d'un tableau. Créer une procédure pour cela permet de pouvoir facilement afficher des polynômes à plusieurs reprises lors d'un algorithme portant sur les polynômes, ceci sans avoir à se préoccuper, ni s'encombrer de la technique se cachant derrière. Dans cette procédure `str` est une fonction convertissant une valeur numérique en chaîne de caractères.

Procédure 3.14 : affichepolynome(T)

Entrée : T : tableau représentant un polynôme
Sortie : ∅

```
ch ← '' ;
pour i ← Taille (T)-1 descendant à 0 faire
    si T[i] ≠ 0 alors
        si (T[i] > 0) alors
            si ch ≠ '' alors
                ch ← ch + ' + '
            fin si
        sinon
            ch ← ch + ' - '
        fin si
        si (|T[i]| ≠ 1) ∨ (i = 0) alors
            ch ← ch + str(|T[i]|);
            si i ≠ 0 alors
                ch ← ch + ' * '
            fin si
        fin si
        si i ≠ 0 alors
            ch ← ch + 'X'
        fin si
        si i > 1 alors
            ch ← ch + '^' + str(i)
        fin si
    fin si
fin pour
si ch = '' alors
    ch ← '0'
fin si
Afficher (ch)
```

Une fonction est similaire à une procédure, mais renvoie en plus une valeur de sortie. En général, il est conseillé de faire en sorte que la seule action d'une fonction soit ce retour d'une valeur. En particulier, on ne fait aucune interface avec l'utilisateur : pas d'affichage, ni de lecture de valeur (les valeurs à utiliser pour la fonction étant alors passées en paramètres).

Par exemple, la fonction 3.15 retourne une valeur entière n. À quoi correspond-elle ?

Définition 3.1.2 (Récursivité)

Une fonction est dite récursive si elle s'appelle elle-même pour une autre valeur des paramètres. Il faut prendre garde à initialiser la récursivité, en donnant explicitement la valeur obtenue pour certaines valeurs des paramètres, en s'assurant que ces valeurs de paramètres sont absorbantes.

La fonction 3.16 est une fonction récursive pour le calcul la suite définie par une récurrence simple, en l'occurrence $u_{n+1} = \sin(u_n)$, initialisée par 1.

La fonction 3.17 est une très mauvaise façon de calculer le n-ième terme de la suite de Fibonacci par récursivité. En pratique, vous calculerez plus vite à la main F_{100} que l'ordinateur par cette fonction. Pourquoi cet algorithme récursif est-il si mauvais ?

Fonction 3.15 : comptea(ch)

Entrée : ch : chaîne de caractères
Sortie : n : entier

$n \leftarrow 0$;
pour $i \leftarrow 0$ **à** `Taille(`ch`)` **faire**
  si ch[i] = *'a'* **alors**
    $n \leftarrow n + 1$
  fin si
fin pour
renvoyer n

Fonction 3.16 : un(n)

Entrée : n : entier positif
Sortie : un(n) : réel

si $n = 0$ **alors**
  renvoyer 1
sinon
  renvoyer $\sin(\text{un}(n-1))$
fin si

Fonction 3.17 : fibo(n)

Entrée : n : entier positif
Sortie : fibo(n) : réel

si $n = 0$ **alors**
  renvoyer 0
sinon si $n = 1$ **alors**
  renvoyer 1
sinon
  renvoyer $\text{fibo}(n-1) + \text{fibo}(n-2)$
fin si

L'étude de la récursivité est du ressort du programme de Spé (ou de Sup en cours d'option). Nous nous contenterons donc cette année d'une utilisation naïve de la récursivité, tout en restant conscient des dangers d'explosion de complexité que cela peut amener.

II Étude de quelques algorithmes de recherche

Ces quelques algorithmes interviennent fréquemment dans des algorithmes plus élaborés. On se contente souvent d'ailleurs d'utiliser des fonctions toutes faites, sans vraiment se préoccuper de ce qui se cache derrière. Il est cependant important de connaître leur fonctionnement, ainsi que leur complexité (afin de pouvoir estimer la complexité des algorithmes qui les utilisent).

II.1 Recherche du maximum d'une liste

Problème : Étant donné une liste de réels, trouver le maximum de cette liste (ou de façon similaire, son minimum).

Idée de résolution : Progresser dans le tableau en mémorisant le plus grand élément rencontré jusque-là. On peut aussi mémoriser sa place.

On utilise la convention usuelle de Python pour les indexations : un tableau de taille n est supposé indexé de 0 à $n-1$.

Algorithme 3.18 : Recherche maximum

Entrée : T, tableau de réels de taille n
Sortie : $\max(T)$
$M \leftarrow T[0]$;
pour $i \leftarrow 1$ **à** $n-1$ **faire**
  si $T[i] > M$ **alors**
    $M \leftarrow T[i]$
  fin si
fin pour
renvoyer (M)

II.2 Recherche d'un élément dans une liste

Problème : Étant donné une liste, trouver un terme particulier dans cette liste, s'il y est.

Idée de résolution : Parcourir la liste jusqu'à ce qu'on trouve l'élément souhaité. Variantes : si on veut toutes les occurrences, on parcourt la liste en entier ; si on veut la dernière occurrence, on parcourt le tableau à partir de la fin.

Algorithme 3.19 : Recherche première occurrence

Entrée : T, tableau de réels de taille n ; a élément à chercher dans T
Sortie : i : indice de la première occurrence de a, ou `None` si pas d'occurrence
$i \leftarrow 0$;
tant que $i < n$ *et* $T[i] \neq a$ **faire**
  $i \leftarrow i+1$
fin tant que
si $i < n$ **alors**
  renvoyer (i)
fin si

Remarquez qu'on utilise ici les tests booléens dans le mode « paresseux », ce qui signifie que pour obtenir la valeur du booléen « b_1 et b_2 », dès lors que b_1 est faux, on ne va pas voir b_2 et on renvoie « faux ». Ainsi, dans cette boucle, si $i = n$, le test $T[i] \neq a$ n'est pas effectué. Sinon, on aurait un problème de validité des indices (on sort du tableau). Beaucoup de langages (dont Python) effectue les opérations booléennes en mode paresseux.

Remarque 3.2.1

Si le tableau T est utilisé essentiellement pour des tests d'appartenance, il faut se demander si la structure de tableau est vraiment adaptée : il est peut-être plus intéressant de considérer une structure d'ensemble, le test d'appartenance s'y faisant plus rapidement.

II.3 Recherche dans une liste triée

Si la liste est triée, rechercher un élément, ou rechercher la position d'un élément à insérer, se fait beaucoup plus rapidement.

Problème : Étant donné un tableau T trié et un élément a comparable aux éléments du tableau, trouver la première occurrence de a (donc pouvoir dire si a est dans le tableau ou non), ou, de façon équivalente, trouver le plus petit indice i tel que $a \leqslant T[i]$ (dans ce cas, a est dans le tableau, de plus petit occurrence i, si et seulement si $a = T[i]$). Ce dernier problème étant un peu plus général, c'est lui qu'on étudie.

Idée 1 : parcourir le tableau dans l'ordre pour repérer i. Cela ne change pas grand chose à la méthode de recherche dans un tableau non trié. En particulier, en moyenne, le coût sera linéaire.

Idée 2 : faire une dichotomie, en coupant à chaque étape le tableau en 2, afin de n'en garder que la moitié pertinente. Intuitivement, cela va beaucoup plus vite, puisqu'à chaque étape, on réduit le nombre de possibilités de moitié.

Algorithme 3.20 : Recherche par dichotomie dans tableau trié

```
Entrée : T, tableau de réels de taille n, trié ; x élément à positionner dans T
Sortie : i : indice minimal tel que T[i] ⩾ x, éventuellement i = n
a ← 0 ;
b ← n − 1 ;
si T[b] < x alors
    renvoyer (n)
sinon si T[a] ⩾ x alors
    renvoyer (0)
sinon
    tant que b − a > 1 faire
        c = ⌊(a+b)/2⌋ ;
        si T(c) ⩾ x alors
            b ← c
        sinon
            a ← c
        fin si
    fin tant que
fin si
renvoyer (b)
```

Exercice 1

1. Expliquer comment améliorer l'algorithme si on ne recherche pas nécessairement la première occurrence.
2. Expliquer comment récupérer la dernière occurrence.
3. Comment récupérer l'ensemble de toutes les occurrences ?
4. Proposer une variante de cet algorithme pour obtenir un test d'appartenance à une liste triée.

Remarque 3.2.2

On prouvera rigoureusement un peu plus tard que cet algorithme s'arrête toujours et fournit bien ce qu'on veut, à l'aide de variants et d'invariants de boucles.

II.4 Autres algorithmes

Les recherches ci-dessus peuvent bien sûr s'adapter à une chaîne de caractères. Un problème fréquent est celui de la recherche d'un mot dans un texte : on peut positionner le mot en toute place du texte, et faire une comparaison lettre à lettre : en cas d'échec, on déplace le mot d'un cran, tant que c'est possible. On se rend bien compte que la complexité est en $O(nk)$, où n est la longueur du texte et k la longueur du mot.

Moyennant un prétraitement du texte (confection d'une table triée des suffixes, pouvant se faire de façon naïve en $\Theta(n\ln(n))$: pour une donnée de taille n, le temps de réponse est de l'ordre de $n\ln(n)$), on peut améliorer les performances de la recherche en descendant à $O(\ln(n))$ (en considérant k comme constante), sans compter le prétraitement. Avec le prétraitement naïf, on y perd (mais on peut faire ce prétraitement

plus rapidement). Ainsi, cela s'avère intéressant si le prétraitement est fait une fois pour toutes (dans un texte finalisé qui n'est pas trop amené à être modifié ; des mises à jours restent possibles), et est antérieur à la demande de recherche. C'est ce type d'algorithmes qui est à la base des recherches dans les pages internet.

On verra en TP quelques algorithmes de tri de tableau. Ces algorithmes sont utiles dans la vie pratique tous les jours (par exemple lorsqu'on ordonne des données dans un tableur ou une base de données). Ils sont aussi à la base de nombreux algorithmes plus sophistiqués. Il est donc important de pouvoir trouver des tris de complexité minimale. Nous verrons plusieurs algorithmes en $\Theta(n^2)$ (tri à bulles, tri par sélection, tri par insertion), et quelques algorithmes en $\Theta(n\ln(n))$ (en moyenne) (tri rapide, tri fusion). On peut montrer que dans une situation générale, c'est le plus rapide qu'on puisse faire. Sous certaines hypothèses supplémentaires (par exemple travailler sur un ensemble fini de données), on peut descendre encore.