Comment aborder l'écriture d'un algorithme ?

Comment organiser cette écriture ?

Jean-Cédric Chappelier (version 1.0), modifiée par Ronan Boulic

Version 1.10 – Juillet 2021

Ce document donne quelques conseils sur la méthode de travail pour se lancer dans l'écriture d'un algorithme puis, dans un second temps, pour l'écrire de façon formelle dans le cadre du cours « Information, Calcul et Communication ».

Dans la première étape nous nous concentrons sur les moyens d'identifier les composantes de la solution d'un problème conduisant à l'*ébauche* de l'algorithme. Dans la seconde étape nous insistons sur le style et la *syntaxe* d'écriture du pseudocode.

La section 5 ne concerne pas le cours ICC du premier semestre. Elle est réservée au projet du cours de Programmation Orientée Projet du second semestre pour lequel nous demandons régulièrement d'écrire du pseudocode. Dans ce cadre il est utile d'élargir la palette des concepts que l'on peut utiliser en pseudocode pour faciliter la conversion de l'algorithme vers un programme en C++.

1 De la donnée du problème à *l'ébauche* de l'algorithme

Le titre de cette section suggère implicitement deux croyances naïves qu'il faut tout d'abord identifier pour faciliter la résolution d'un problème :

- **Mythe de la donnée univoque** :

Une donnée univoque pourrait se traduire quasiment mot à mot en un programme car elle ne contient aucune ambiguïté. Malheureusement, la donnée étant écrite en langage naturel, il existe très souvent des possibilités d'interprétations différentes entre ce qu'a voulu exprimer la personne qui l'a rédigée et ce qu'a compris la personne qui la lit. C'est pourquoi il est tout à fait normal de poser des questions sur la manière d'interpréter la donnée d'un problème.

- **Mythe de la solution unique** :

La majorité des problèmes accepte une grande variété de solutions, certaines étant plus efficaces que d'autres. Nous recommandons de ne pas hésiter à explorer plusieurs voies dans une première phase pour ensuite vous concentrer sur celle qui présente les meilleurs atouts. Plus difficile : il faut garder l'esprit ouvert et savoir *accepter de remettre en question* une approche sur laquelle on a déjà investi du temps car finalement elle est peu performante.

1.1 Les premières questions à se poser

Le tout premier conseil pour résoudre un problème est de ne pas commencer par la syntaxe (« *comment* écrire? ») mais, vraiment, de commencer par le fond/le but (« *quoi* écrire? ») : ne vous bloquez pas sur comment écrire votre algorithme si vous ne savez pas encore clairement ce que vous voulez écrire.

Le premier conseil est donc de réfléchir, faire un/des brouillon(s), schémas, etc. Cette première étape est essentiellement une étape ***papier-crayon*** (ou tablette/stylet) dans laquelle on travaille la matière de la donnée de façon à identifier:

- l'idée d'une méthode systématique reformulant les éléments de la donnée

En l'absence d'une idée claire dès le départ, on cherchera à verbaliser :

- les sous-problèmes ou actions à effectuer

- la séquence temporelle dans lequel ces actions doivent se succéder

- les éventuelles conditions auxquelles une action, ou sa répétition, est soumise

- sous quelle forme est-il préférable d'organiser les données pour être plus efficace

Exemples : le cours fournit plusieurs illustrations de cette approche descendante en montrant, indépendamment de la donnée, des questions/réponses clarifiant le problème (M1.L2 slide 28), le principe de la solution (M1L2 slides 30-35, M1.L4 slide 9, 40), une ébauche de haut-niveau (M1.L3 slide 54, M1.L4 slides 27, 29).

1.2 Le rôle important de la structuration des données

En matière de structuration des données, à coût égal en matière d'espace mémoire, de calcul et de lisibilité de l'algorithme, on cherchera en priorité à mettre en oeuvre le concept le plus simple possible. C'est pourquoi les sous-sections suivantes présentent d'abord les approches les plus simples à privilégier.

1.2.1 Eviter de calculer plusieurs fois la même chose

Le cours M1.L4 valorise le fait de ***mémoriser des calculs intermédiaires*** pour éviter de les refaire encore et encore. Cette stratégie est au cœur de l'approche appelée « programmation dynamique ».

Dans certains cas une mémorisation pertinente peut faire diminuer l'ordre de complexité dans la résolution d'un problème (ex : dans M1.L4 le calcul du triangle de Pascal ou de la suite de Fibonacci).

1.2.2 Variable, Liste, Table

Le cours a introduit essentiellement deux outils pour l'organisation des données :

- la ***variable*** mémorise une seule donnée
- la ***liste*** mémorise un ensemble de données auxquelles on accède avec un indice entier i compris entre 1 et la taille de la liste N. Le premier élément de la liste L est L(1) et le dernier est L(N).

La notion de liste peut être étendue en liste à plusieurs indices pour traiter des entités telles que des images, matrices etc. On parle souvent de ***table*** pour une liste à deux indices. Par exemple en M1.L4 on travaille avec une table des distances D à deux indices. On peut noter un élément de cette table D avec **D(i)(j)** ou **D(i,j)** avec :

- l'indice **i** à **gauche** est l'indice de ***ligne***
- l'indice **j** à **droite** est l'indice de ***colonne***

(pour se souvenir, penser au mot **licol**).

1.2.3 Comment rassembler plusieurs données de types différents

Si l'entité X manipulée par l'algorithme est composée de plusieurs données de natures différentes (ex : entier, nombre à virgule, booléen, chaine de caractères, etc…), celles-ci ne peuvent pas appartenir à la même liste car les éléments d'une liste doivent tous être de même nature.

La première solution n'introduit aucune nouveauté par rapport aux conventions précédentes. Il suffit d'avoir autant de variables indépendantes qu'il y a de données de nature différente pour représenter X. Si on veut travailler avec un ensemble de N entités de même nature que X, alors on peut créer une liste de taille N pour chacune des données qui composent X (ex : une liste de taille N de noms, une autre d'âges, une autres de salaires, etc…)

La seconde solution correspond au concept de *structure* en programmation. Chaque donnée composant l'entité X possède un nom de variable. On associe ce nom de variable a l'entité X en les reliant avec l'opérateur point (ex : X.nom , X.age, etc…). On peut ensuite travailler avec une liste de N entités de même nature que X et accéder à la variable de l'élément d'indice i (ex : L(i).nom, L(j).age, etc).

1.2.4 Comment représenter la notion de lien entre deux éléments ?

Certaines structures de données élaborées comme des graphes ont besoin d'indiquer qu'un élément est relié à un autre élément, en bref qu'il existe un *lien* entre ces deux éléments.

Le moyen classique utilisé en pseudocode pour représenter ce lien est très simple : il suffit de représenter l'ensemble des éléments par une liste de taille N. Chaque élément de la liste est désigné par son indice i dans la liste. C'est cette valeur d'indice i qu'il suffit de mémoriser pour exprimer un lien.

Si un élément est lié à plusieurs autres éléments il suffit de mémoriser une liste d'indices.

Remarque : l'implémentation de la notion de *lien* dans un langage donnée peut tirer parti d'autres concepts qu'on n'utilise pas en général dans du pseudocode. Par exemple *l'adresse mémoire* d'un élément de l'algorithme. Cependant, explicitement utiliser *l'adresse mémoire* d'un élément d'un algorithme est rarement une bonne idée dans la conception d'un algorithme mais parfois on ne peut pas faire autrement.

La bonne pratique est justement de n'utiliser la notion d'adresse mémoire d'un élément **SEULEMENT quand on ne peut pas faire autrement.** Dans ce cas, il est possible d'utiliser les opérateurs & pour déterminer l'adresse d'une entité et * pour obtenir la valeur rangée en mémoire à une adresse mémorisée dans un pointeur.

La section 5 formalise ce qui est autorisé dans le cadre du projet de programmation orientée projet du second semestre.

La page suivante est optionelle ; elle présente brièvement des concepts fréquemment utilisés en informatique pour organiser des données associées à un usage bien précis (pile, file d'attente)

1.2.5 Pile *(optionel)*

La notion de ***pile*** n'est pas traitée dans le cours théorique mais mérite de l'être ici car elle se traduit très facilement en programmation (avec l'outil de ***vector***). Une pile est comme une liste à un seul indice à la différence qu'**on ne peut accéder qu'à un seul élément de cette liste = le dernier qu'on y a rangé (LIFO = Last In, First Out),** comme dans une pile d'assiette où on ne peut reprendre que la dernière qu'on a posé sur la pile. Initialement une pile est vide et on peut y ajouter un élément à la fois. Celui-ci est mis à la suite du dernier élément de la pile et devient le nouveau « dernier élément de la pile ». Si la pile n'est pas vide, on ne peut enlever qu'un seul élément à la fois : c'est le « dernier élément » actuel. Les fonctions autorisées sur une Pile P sont les suivants :

Taille(P) : renvoie la taille de la pile P (éventuellement nulle)
Lire(P) : renvoie la valeur du dernier élément de la Pile P mais sans l'enlever
Extraire(P) : enlève et renvoie la valeur du dernier élément de la Pile P
Ajouter(P, *valeur*) : ajoute *valeur* sur le dessus de la Pile P

1.2.6 File d'attente *(optionel avancé)*

La notion de ***file d'attente*** (***queue*** en anglais) n'est pas traitée dans le cours théorique ni en C++ au premier semestre. Une file d'attente est comme une liste à un seul indice à la différence qu'**on ne peut accéder qu'à un seul élément de cette liste = le premier qu'on y a rangé (FIFO = First In, First Out),** comme dans une file d'attente à un guichet. Initialement une file est vide et on peut y ajouter un élément à la fois. Celui-ci est mis à la suite du dernier élément de la file et devient le nouveau « dernier élément de la file ». Si la file n'est pas vide, on ne peut enlever qu'un seul élément à la fois : c'est le « premier élément » qui y a été entré. Les fonctions autorisées sur une File Q sont les suivants :

Taille(Q) : renvoie la taille de la file Q (éventuellement nulle)
Lire(Q) : renvoie la valeur du premier élément de la file Q mais sans l'enlever
Extraire(Q) : enlève et renvoie la valeur du premier élément de la file Q
Ajouter(Q, *valeur*) : ajoute *valeur* comme dernier élément de la file Q

1.2.7 Combinaison de Pile et File d'attente *(optionel très avancé)*

Pour votre culture générale, il est possible de combiner les actions effectuées sur une pile et une file d'attente dans une même structure de donnée appelée ***deque*** qui est l'abréviation de ***double ended queue*** (en programmation C++). On précise le coté (**front** ou **back**) dans le nom des fonctions qui agissent sur une telle entité.

2 De l'ébauche de l'algorithme au pseudocode

Une fois au clair sur le « quoi », et seulement à ce moment là, préoccupez-vous de la mise en forme.

Commencez pour cela par écrire formellement (en français tout de même) la description la plus précise possible **des entrées fournies à l'algorithme** et **la sortie** obtenue.

Au niveau des **entrées**, pour avoir une correspondance plus directe avec les langages de programmation, on distingue deux familles pour lesquelles on adopte la définition suivante:

- **Entrée = Entrée non-modifiée**: les valeurs fournies à l'algorithme peuvent être modifiée dans l'algorithme MAIS *ces modifications n'ont aucune conséquence à l'extérieur de l'algorithme.* C'est le cas classique et recommandé pour la robustesse de l'algorithme mais il induit parfois un surcoût du fait de la copie des valeurs des paramètres.

- **Entrée modifiée** : les entités fournissant ces valeurs en entrées peuvent être modifiées dans l'algorithme et cela *se répercute sur les entités externes à l'algorithme qui sont ainsi modifiées.* Cela correspond au passage par référence dans certains langages.

L'ordre des paramètres dans le prototype de la fonction devrait toujours refléter le même ordre: **entrées non modifiées / entrées modifiées** (on privilégie cet ordre car il est le plus proche de l'ordre **entrées / sorties**).

Remarque : ne pas confondre un *affichage* avec une « sortie ». Il est rare qu'un algorithme mentionne l'action d'afficher sauf si c'est explicitement le problème à résoudre.

Exemple : algorithme de recherche d'*une* des valeurs maximales dans une liste :

Valeur maximale
entrée : L *une liste non vide de nombres* **sortie** : *la (ou une des) valeur(s) maximale(s) de la liste*

Par défaut, l'indication « **entrée** », utilisée seule, veut dire **entrée non-modifiée**
Par défaut, l'indication « **sortie** » doit être associée à un résultat transmis avec l'instruction **Sortir** indiquée explicitement dans l'algorithme (expliquée plus loin).

3 Les composantes du pseudocode

Utilisez ensuite les instructions suivantes pour écrire votre pseudocode :

— affectation : ←
p.ex. : $x \leftarrow 3$

— toutes les opérations mathématiques : notation usuelle
p.ex. : $x \geq 2$

— désignation d'un élément d'une liste : parenthèses rondes () ou carrées [], au choix
p.ex. : le i-ème élément de la liste L : $L(i)$ ou $L[i]$

3.1 Les structures de contrôle.

TRES important : décaler sur la droite les instructions contrôlées (**indentation**)
On peut ajouter une barre verticale pour mieux les distinguer

3.1.1 « **Si** ... alors ... Sinon …» :

Le mot « alors » n'est pas écrit pour alléger le pseudocode
La clause **Sinon** est optionnelle

Si condition
    Instructions

Si condition
    Instructions
Sinon
    Instructions

On remplacera l'écriture ci-dessous par celle-ci :

Si condition
    // ne rien faire
Sinon
    Instructions

Si négation de la condition
    Instructions

3.1.2 « **Tant que** ... » : la **boucle conditionnelle** possède deux formes

Tant que condition
    Instructions

Faire
    Instructions
Tant que condition

3.1.3 « **Pour …** » encore appelée **itération**

La boucle **Pour** à la forme suivante :

Pour *i* de *valeur_initiale* à *valeur_finale*
Instructions

Les conventions concernant les boucles « **Pour** » incluent :

- la *valeur_initiale* n'est utilisée qu'*une seule fois* pour initialiser la variable de boucle *i*, *avant* le premier passage dans la boucle.
- la *valeur_finale* peut éventuellement être modifiée par les instructions contrôlées
- Si *valeur_initiale* ≤ *valeur_finale* alors l'incrément est implicitement de 1
- Si *valeur_initiale* ≥ *valeur_finale* alors l'incrément est implicitement de -1
- En dehors de ces cas il faut préciser explicitement la valeur de l'incrément;

  Pour *i* allant de 1 à *n* de 2 en 2 // *ou* : par pas de 2
  Instructions

- En cas d'ambiguïté, si une ou les deux bornes sont des variables qui *peuvent prendre des valeurs autorisant les deux interprétations d'incrément croissant ou décroissant*, il faut explicitement préciser dans quel sens doit varier la variable de boucle.

  Dans l'exemple ci-dessous si *p* peut être aussi bien négatif que positif alors il faut explicitement préciser l'incrément :

  Pour k allant de p à 0 par pas de -1
  Instructions

Si l'ensemble décrit par la boucle est l'ensemble vide, la boucle ne se déroule pas du tout; p.ex.

Pour *i* allant de 1 à n par pas de 1
Instructions

ne fera *rien* si *n* est inférieur ou égal à 0.

On utilisera la boucle conditionnelle **Tant que** si une *condition* supplémentaire doit être remplie pour poursuivre l'itération.

3.1.4 « **Pour tout** … » :

Cette formulation de boucle peut être utilisée s'il n'est pas nécessaire de connaître la position/l'indice de l'élément dans l'ensemble *L* pour traiter cet élément :

Pour tout élément x de L

Instructions utilisant seulement *x*

Remarque : en pseudocode on considère que l'élément x de L est modifié lorsqu'on écrit des affection du type : x ← … . Si on ne veut pas modifier l'élément x de L, alors il faut initialiser une autre variable avec la valeur de x et faire les modifications sur cette autre variable.

3.1.5 Terminaison prématurée d'une boucle ou itération

La terminaison prématurée d'une boucle doit être associée au test d'une condition ; elle peut être exprimée avec l'instruction inconditionnelle suivante :

```
Pour …..
    Instructions
    Si condition
        Sortir de la boucle
    Instructions
```

3.1.6 Passage prématuré à l'itération suivante d'une boucle

Certains langages permettent de *passer prématurément à l'itération suivante d'une boucle* à l'aide d'un mot clef comme par exemple « continue » associé au test d'une expression logique. Cependant, il suffit d'inverser le test pour obtenir le même résultat. C'est pourquoi nous recommandons de remplacer la forme de gauche par celle de droite:

```
Pour …..                          Pour …
    Instructions                      Instructions
    Si condition                      Si négation de la condition
        Continue                          Instructions_suivantes
    Instructions_suivantes
```

3.1.7 Bonne pratique sur le point de sortie

Il est recommandé d'avoir un seul point de sortie d'une boucle, celui qui est exprimé au début avec le mot-clef **Tant que** ou implicitement avec la borne finale d'une itération **Pour**.

S'il y a des conditions supplémentaires à remplir il est préférable de les rassembler en ce point de sortie unique. La boucle **Tant que** est plus adaptée que l'itération **Pour** pour exprimer cette souplesse supplémentaire de l'exécution.

L'approche 3.1.5 est néanmoins acceptée pour traiter des conditions additionnelles.

3.1.8 Terminaison de l'algorithme ou d'une fonction : « **Sortir** : »

Sortir : x

Notez que l'instruction « Sortir : » met fin à l'algorithme (même s'il y a encore des lignes en dessous). S'il s'agit d'une fonction cela quitte la fonction en retournant éventuellement une valeur utilisée au niveau supérieur de l'algorithme.

3.2 Actions de plus haut niveau.

— si nécessaire, pour afficher une valeur/expression, utilisez simplement « afficher »; p.ex.:
afficher x.

Sauf mention contraire dans la donnée, vous pouvez également utiliser tout algorithme *vu en cours* (taille, tri, recherche, plus court chemin) en le désignant par un nom suffisamment clair; par exemple :
— $n \leftarrow \text{taille}(L)$

— $L^0 \leftarrow \text{trier}(L)$ ou $L^0 \leftarrow \text{tri}(L)$

Notes :

Au niveau formel, il est préférable de considérer que les algorithmes ne modifient pas leur entrée mais produisent un nouvel objet (comme une fonction mathématique). Par exemple ci-dessus, la liste *L* n'est pas modifiée par l'algorithme de tri, mais celui-ci retourne une nouvelle liste (triée). Le cours M1.L4 illustre ce point.

Quand on ré-utilise un algorithme que l'on n'a pas écrit soi-même, on précisera par précaution la nature des paramètres d'entrée et s'ils sont modifiés ou pas.

4 Exemple

Voici l'exemple complet, la recherche d'*une* des valeurs maximales dans une liste :

Valeur maximale
entrée : L *une liste non vide de nombres* **sortie** : *la (ou une des) valeur(s) maximale(s) de la liste*

$n \leftarrow \text{taille}(L)$
$x_{max} \leftarrow L(1)$
Pour i allant de 2 à n
  Si $L(i) > x_{max}$
  $x_{max} \leftarrow L(i)$
Sortir : x_{max}

Notez que l'algorithme ci-dessus est correct dans tous les cas en raison des conventions :

— la boucle « **Pour** » ne fait rien si n vaut 1 (et donc, dans ce cas, on retourne finalement $L(1)$);

— la description de l'entrée est toujours vraie : ci-dessus la liste L ne peut (axiomatiquement) pas être vide; il est donc important de bien préciser les hypothèses de départ. Par exemple l'algorithme suivant n'est pas correct :

Valeur maximale
entrée : L *une liste de nombres* **sortie** : *la (ou une des) valeur(s) maximale(s) de la liste*
$n \leftarrow taille(L)$ $x_{max} \leftarrow L(1)$ *etc.*

car $L(1)$ n'est pas défini pour une liste vide (et que l'on n'a pas empêché cette possibilité *a priori*). Il faut donc l'écrire comme donné plus haut, et pas autrement, car il n'y a de toutes façons pas de définition de « la valeur maximale » pour une liste vide.

5 Extensions pour le sem2 (projet du cours de Programmation Orientée Projet)

Nous demandons régulièrement d'écrire du pseudocode dans des rendus du projet C++ du second semestre. Or, comme le cours de C++ du sem2 introduit plusieurs concepts de l'Orienté Objet, il est donc logique que l'écriture de pseudocode s'aligne avec ces nouveautés.

5.1 Usage de classe, attributs et méthodes.

Tout d'abord la structuration des données peut faire appel à des *classes* que l'on peut voir comme une extension naturelle de la notion de *structure* (1.2.3). La syntaxe d'accès aux données (attributs) s'aligne sur celle des structures avec l'opérateur point.

L'accès aux attributs n'est permis que dans la portée de la classe.

De même il est possible de décrire des fonctions membres d'une classe (*méthode*). Dans ce cas, il faut s'assurer que la personne qui lit le pseudocode a accès à une description des attributs auxquels la méthode peut accéder implicitement.

5.2 Usage de l'adresse d'une entité

La bonne pratique reste de n'utiliser la notion d'adresse mémoire d'un élément SEULEMENT quand on ne peut pas faire autrement. Ce cas de figure est cependant plus fréquent au second semestre.

Dans ce cas, il est possible d'utiliser l'opérateur & pour déterminer l'adresse d'une entité et l'opérateur * pour obtenir la valeur rangée en mémoire à une adresse mémorisée dans un pointeur.

5.3 Interface de vector

L'usage de l'entité ***vector*** en C++ étant très fréquent au second semestre, on autorise de ré-utiliser telle quelle son interface (noms des méthodes) dans du pseudocode.