Introduction à l’Algorithmie

Table des matières

Introduction

Qu'est-ce que l'algorithmie ?

Considérons les étapes qui interviennent dans la résolution problème quelconque :

1. Concevoir une procédure qui une à fois appliquée amènera à une solution du problème;
2. Résoudre effectivement le problème en appliquant cette méthode.

Le résultat du premier point sera nommé un *algorithme*. Quant au deuxième point, c'est-à-dire la mise en pratique de l'algorithme, nous l'appellerons un *processus*. Ces notions sont très répandues dans la vie courante. Un algorithme peut par exemple y prendre la forme :

- D'une recette de cuisine,
- D'un mode d'emploi,
- D'une notice de montage,
- D'une partition musicale,
- D'un texte de loi,
- D'un itinéraire routier.

Dans le cas particulier de l'informatique, une étape supplémentaire vient se glisser entre la conception de l'algorithme et sa réalisation à travers un processus : l'algorithme doit être rendu compréhensible par la machine que nous allons utiliser pour résoudre effectivement le problème. Le résultat de la traduction de l'algorithme dans un langage connu de la machine est appelé un *programme*.

Un peu d'histoire

C'est un mathématicien perse du 8ème siècle, Al-Khawarizmi, qui a donné son nom à la notion d'algorithme. Son besoin était de traduire un livre de mathématiques venu d'Inde pour que les résultats et les méthodes exposés dans ce livre se répandent dans le monde arabe puis en Europe. Les résultats devaient donc être compréhensibles par tout autre mathématicien et les méthodes applicables, sans ambiguïté. En particulier, ce livre utilisait une numérotation de position, ainsi que le chiffre zéro que ce type de représentation des nombres rend nécessaire. Par ailleurs, le titre de ce livre devait être repris pour désigner une branche des mathématiques, l'algèbre.

Si l'origine du mot *algorithme* est très ancienne, la notion même d'algorithme l'est plus encore. On la sait présente chez les babyloniens, 1800 ans avant JC. Une tablette assyrienne provenant de la bibliothèque d'Assurbanipal et datée de -640 expose une recette pour obtenir des pigments bleus (notons que pour avoir un algorithme satisfaisant, il faudrait préciser ici les temps de cuisson !) :

```
tu ajouteras à un mana de bon tersitu un tiers de mana de verre sirsu broyé,
un tiers de mana de sable, cinq kisal de craie, tu broieras encore,
tu le recueilleras dans un moule, en le fermant avec un moule réplique,
tu le placeras dans les ouvreaux du four, il rougeoira et uknû merku en sortira.
```

En résumé, il doit être bien clair que cette notion d'algorithme dépasse, de loin, l'informatique et les ordinateurs. Il nécessite un vocabulaire partagé, des opérations de base maîtrisées par tous et de la précision.

Vers un langage descriptif des algorithmes

Eléments de base

Instructions de sorties

On se donne une instruction *Écrire* pour afficher du texte, ainsi qu'une instruction *ALaLigne* pour poursuivre l'affichage à la ligne suivante.

Patron d'un algorithme

Le patron général est le suivant :

```
Algorithme NomAlgorithme
Début
... actions
Fin
```

La première ligne, appelée *profil* ou *prototype* donne essentiellement le nom de l'algorithme. On trouve ensuite un délimiteur de début puis les différentes actions composant l'algorithme et enfin un délimiteur de fin. Ainsi, l'algorithme suivant est valide :

```
Algorithme Bonjour
Début
  Écrire('Hello world !!!')
  ALaLigne
Fin
```

Variables et types

Une variable est constituée d'un nom et d'un contenu, ce contenu étant d'un certain type. Les types différents : booléen, caractère, chaîne de caractères, nombre entier, nombre réel, etc.

Pour la clarté de l'algorithme, il peut être intéressant de déclarer les variables utilisées et leur type au tout début. Quand l'algorithme sera traduit en programme cette déclaration aura d'autres avantages : réservation de l'espace mémoire correspondant au type, possibilité de vérifier le programme du point de vue de la cohérence des types, etc.

Par tradition, on notera l'affectation d'une valeur à une variable en utilisant le symbole flèche vers la gauche (←).

```
Algorithme Valeur3
  X : Entier
Début
  X ← 3
  Écrire(X)
  ALaLigne
Fin
```

Nouveau patron d'un algorithme

Nous allons maintenant préciser les variables utilisées par l'algorithme et leurs types, en distinguant les paramètres externes et les variables internes à l'algorithme. Ainsi, le patron d'un algorithme devient :

```
Algorithme NomAlgorithme (paramètres...)
Variable ...
Début
... actions
Fin
```

Instructions conditionnelles et itératives

Un certain nombre de tâches nécessite de répéter un groupe d'actions soit un nombre de fois donné, soit jusqu'à qu'un événement ou une condition soit rempli. D'autre cas nécessite de pouvoir choisir un comportement parmi plusieurs en fonction d'évènement ou de conditions.

Structure conditionelle : Si-Alors-Sinon

Démarrons par un exemple :

```
Algorithme QueFaireCeSoir
Début
  Si Pluie
  Alors
     MangerPlateauTélé
     SeCoucherTot
  Sinon
     MangerAuRestaurant
     AllerAuCinema
  Fin si
Fin
```

La structure Si-Alors-Sinon permet de tester une condition (ici 'Pluie'). Si cette condition s'avère remplie (il pleut effectivement) , alors les actions contenus dans le bloc Alors sont exécutées. Si elle s'avère fausse (pas de pluie à l'horizon), alors les actions du bloc Sinon sont exécutées.

Le bloc Sinon est optionnel. Dans ce cas, si la condition n'est pas remplie, aucune action n'est exécutée.

Boucle Tant que et Répéter

Les boucles Tant Que et Répéter vont exécuter une liste d'actions jusqu'à ce qu'une condition ne soit plus remplie.

```
Algorithme JusquAuMur
Début
  Tant que Non(DevantMur) faire
    Avancer
  Fin tant que
Fin

Algorithme JusquAuMurVersionRépéter
Début
  Répéter
    Avancer
  jusqu'à DevantMur
Fin
```

À noter que, contrairement à la boucle *tant que*, on passe toujours au moins une fois dans une boucle *répéter*. Ainsi, dans l'exemple ci-dessus, on avancera forcément d'une case. Il conviendrait donc de tester si l'on n'est pas devant un mur avant d'utiliser cet algorithme.

Boucle Pour

Dotés des variables, nous pouvons maintenant écrire un nouveau type de boucle, la boucle *Pour* :

```
Algorithme CompteJusqueCent
Variable i : entier
Début
  Pour i ← 1 à 100 faire
    Écrire(i)
    ALaLigne
  Fin Pour
Fin
```

Lorsque l'on sait exactement combien de fois on doit itérer un traitement, c'est l'utilisation de cette boucle qui doit être privilégiée.

Par exemple,

```
Algorithme DessineEtoiles (n : entier)
Variable i : entier
Début
  Pour i ← 1 à n faire
    Écrire('*')
  Fin pour
Fin
```

Comparaisons des boucles pour un problème simple

On reprend l'exemple précédent de la boucle *Pour :*

```
Algorithme CompteJusqueCentVersionPour
Variable i : entier
Début
  Pour i ← 1 à 100 faire
    Écrire(i)
    ALaLigne
  Fin Pour
Fin
```

On écrit des algorithmes qui produisent la même sortie (les nombres de 1 à 100) mais en utilisant les différentes boucles.

D'abord, avec la boucle *tant que* (il faut initialiser *i* avant la boucle, et l'augmenter de 1 à chaque passage) :

```
Algorithme CompteJusqueCentVersionTQ
Variable i : entier
Début
  i ← 1
  Tant que (i ≤ 100) faire
    Écrire(i)
    ALaLigne
    i ← i+1
  Fin tant que
Fin
```

De même avec la boucle *répéter* (noter que la condition d'arrêt est ici la négation de la condition du *tant que*):

```
Algorithme CompteJusqueCentVersionRepeter
Variable i : entier
Début
  i ← 1
  Répéter
    Écrire(i)
    ALaLigne
    i ← i+1
  Jusqu'à (i > 100)
Fin
```

Enfin, en utilisant la récursivité, on obtient :

```
Algorithme CompteJusqueCentRecursif (n : entier)
Début
  Si (n ≤ 100)
  Alors
    Écrire(n)
    ALaLigne
    CompteJusqueCentRecursif(n+1)
  Fin Si
Fin
```

Types abstraits

Formalisme

On définit un type abstrait comme une spécification mathématique d'un ensemble de données et de l'ensemble des opérations qu'elles peuvent effectuer. On qualifie d'abstrait ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter.

Un type abstrait est composé de cinq champs :

- Type abstrait
- Utilise
- Opérations
- Pré-conditions
- Axiomes

Ces cinq éléments sont souvent réunis sous l'acronyme : TUOPA.

Type abstrait

Le champ « Type abstrait » contient le nom du type que l'on est en train de décrire et précise éventuellement si celui-ci n'est pas une extension d'un autre type abstrait. Par exemple, on écrira « Type abstrait : Pile » pour créer un type nommé Pile qui décrit ce qu'est une pile et « Extension Type abstrait : Pile » si on désire l'étendre (on étend un type abstrait en lui assignant de nouvelles opérations non définies lors de la première spécification).

Utilise

Le champ « Utilise » contient les types abstraits que l'on va utiliser dans celui que l'on est en train de décrire. Par exemple, le type abstrait Pile que l'on définit va utiliser dans sa spécification le type abstrait Booléen, et on écrira « Utilise : Booléen ».

Opérations

Le champ « Opérations » contient le prototypage de toutes les opérations. Par prototypage, on entend une description des opérations par leur nom, leurs arguments et leur retour.

Les opérations sont divisées en plusieurs types :

- Les constructeurs (permettent de créer un objet du type que l'on est en train de définir)
- Les transformateurs (permettent de modifier les objets et leur contenu)
- Les observateurs (fonction donnant des informations sur l'état de l'objet)

Exemple d'opération pour le type « Type abstrait : Pile » :

$$\text{créer} : \emptyset \rightarrow \text{Pile}$$

Notez que l'opération « créer » est un constructeur. En effet, cette fonction crée un objet de type pile. De plus, elle n'a pas besoin d'argument pour créer cette pile. Ceci est montré par l'absence d'indication à gauche de la flèche ou l'utilisation du symbole ensemble vide ($\emptyset$).

Pré-conditions

Le champ « Pré-conditions » contient les conditions à respecter sur les arguments des opérations pour que celles-ci puissent avoir un comportement normal. On peut parler ici d'ensemble de définition des opérations.

Axiomes

Le champ « Axiomes » contient une série d'axiomes pour décrire le comportement de chaque opération d'un type abstrait. Chaque axiome est une proposition logique vraie.

Le type abstrait « Entier »

```
Type abstrait « Entier »
Utilise « Booléen »

Constructeurs
zero : ∅ → Entier
succ : Entier → Entier
prec : Entier → Entier // pré-condition pour prec(n) : estnul(n) est faux

Opérations :
estnul : Entier → Booléen
afficheentier : Entier → ∅

Axiomes
estnul(zero()) = vrai
estnul(succ(n)) = faux
succ(prec(n)) = n
prec(succ(n)) = n
```

Avec ce type abstrait, nous sommes capables de définir l'addition et la multiplication, indépendamment de l'implémentation de ce type. Pour la simplicité de l'écriture, on se contentera de traiter les entiers positifs ou nuls.

```
Algorithme plus (n,m : entiers)
Début
  Tant que non(est_nul(n))
    m ← succ(m)
    n ← prec(n)
  Fin TQ
  Retourner m
Fin

Algorithme fois (n,m : entiers)
Début
  s ← zero()
  Tant que non(est_nul(n))
    s ← plus(s,m)
    n ← prec(n)
  Fin TQ
  Retourner s
Fin
```

Le type abstrait « Pile »

```
Type abstrait « Pile »
Utilise « Booléen » et « Élément »

Constructeurs
pilevide : ∅ → Pile
ajoute : Élément × Pile → Pile

Opérations
sommet : Pile → Élément // pré-condition pour sommet(p) : estvide(p) est faux
depile : Pile → Pile    // pré-condition pour depile(p) : estvide(p) est faux
estvide : Pile → Booléen
affichepile : Pile → ∅
copiepile : Pile → Pile

Axiomes
estvide(pilevide()) = vrai
estvide(ajoute(e,p)) = faux
sommet(ajoute(e,p)) = e
depile(ajoute(e,p)) = p
```

Un algorithme possible basé sur ce type :

```
Algorithme inversepile (p : Pile)
Début
  q ← copiepile(p)
  r ← pilevide()
  Tant que non(estvide(q))
    ajoute(sommet(q),r)
    depile(q)
  Fin TQ
Retourner r
Fin
```

Types *produits*

Le type produit de deux types A et B est l'analogue en théorie des types du produit cartésien ensembliste et est noté A × B. C'est le type des couples dont la première composante est de type A et la seconde de type B. Cette notion s'étend à un nombre de type de base arbitraires et correspond à la notion de **structure** en langage de programmation.

```
Type abstrait « Étudiant »

Utilise « Entier » et « Texte »

Constructeurs
créer_étudiant : Texte × Texte × Texte → Étudiant

Opérations
nom_étudiant : Étudiant → Texte
prénom_étudiant : Étudiant → Texte
naissance_étudiant : Étudiant → Texte
noteinfo_étudiant : Étudiant → Entier
notemath_étudiant : Étudiant → Entier
notediscipline_étudiant : Texte × Étudiant → Entier
modifier_noteinfo : Étudiant × Entier → Étudiant
modifier_notemath : Étudiant × Entier → Étudiant
afficher_étudiant : Étudiant → ∅

Axiomes
nom_étudiant(créer_étudiant(n,p,d)) = n
prénom_étudiant(créer_étudiant(n,p,d)) = p
naissance_étudiant(créer_étudiant(n,p,d)) = d
noteinfo_étudiant(modifier_noteinfo(e,n)) = n
notemath_étudiant(modifier_notemath(e,n)) = n
notediscipline_étudiant('info',modifier_noteinfo(e,n)) = n
notediscipline_étudiant('math',modifier_notemath(e,n)) = n
```

Le type abstrait « Tableau »

```
Type abstrait « Tableau »

Utilise « Entier » et « Élément »

Constructeurs
créer_tableau : Entier → Tableau

Operations
affichetableau : Tableau → ∅
copietableau : Tableau → Tableau
contenu : Tableau × Entier → Élément
taille : Tableau → Entier // pré-condition pour contenu(t,n) : 1 ≤ n ≤ taille(t)

fixecase : Tableau × Entier × Élément → Tableau
// pré-condition pour fixecase(t,n,e) : 1 ≤ n ≤ taille(t)


Axiomes
taille(créer_tableau(n)) = n
contenu(fixecase(t,n,e),n) = e
```

Algorithmes sur les tableaux

Algorithmes de base

Parcours d'un tableau

```
Algorithme AfficheTableau (t : tableau)
Variable i : entier
Début
  Pour i ← 1 à taille(t) faire
  Écrire(t[i])
  Fin Pour
Fin
```

Recherche des plus petit et grand éléments d'un tableau

```
Algorithme Maximum (t : tableau d'entiers)
Variables i, max : entier
Début
  max ← t[1]
  Pour i ← 2 à taille(t) faire
    Si (t[i] > max)
    Alors max ← t[i]
    Fin si
  Fin Pour
  Écrire(max)
Fin
```

On cherche maintenant la position du minimum dans un tableau et entre deux cases repérées par leurs numéros *d* (début) et *f* (fin):

```
Algorithme PositionMinimum (t : tableau d'entiers ; d,f : entier)
Variable i,imin : entier
Début
  imin ← d
  Pour i ← d+1 à f faire :
    Si t[i] ≤ t[imin]
    Alors imin ← i
    Fin si
  Fin pour
  Retourner imin
Fin
```

Existence d'un élément dans un tableau

Algorithme général :

```
Algorithme Recherche (e : entier ; t : tableau d'entiers)
Variable i : entier
Début
  i ← 1;
  Tant que (i ≤ taille(t)) et (t[i] ≠ e) faire
    i ← i+1
  Fin tant que
  Si (i>taille(t))
  Alors Écrire("l'élément recherché n'est pas présent")
  Sinon Écrire("l'élément recherché a été découvert")
  Fin si
Fin
```

Mais si les éléments du tableau sont ordonnés, nous pour vous pouvons tirer parti de cette caractéristique.

```
Algorithme RechercheOrdonnée (e : entier ; t : tableau d'entiers)
Variable i : entier
trouve : booléen
Début
  i ← 1
  Tant que (i ≤ taille(t)) et (t[i] < e) faire:
    i ← i+1
  Fin TQ
  Si (i ≤ taille(t)) et (t[i]=e) alors
    Trouve ← VRAI
  Sinon
    Trouve ← FAUX
  Fin si
  Retourner trouve
Fin
```

Encore mieux avec une recherche dite *dichotomique* :

```
Algorithme RechercheDichotomique(e : entier ; t : tableau d'entiers)
Variable g,d,m : entier
trouve : booléen
Début
  g ← 1
  d ← taille(t)
  trouve ← faux
  Tant que (g ≤ d) et NON(trouve) Faire
    m = (g+d) div 2
    Si t[m] = e
    Alors trouve ← vrai
    Sinon
      Si e < t[m]
      Alors d ← m-1
      Sinon g ← m+1
      Fin si
    Fin si
  Fin Tant que

  Si trouve
  Alors Écrire('Trouvé !')
  Sinon Écrire('Pas trouvé...')
  Fin si

  Écrire(m)
Fin
```

Pour continuer à bénéficier de ces algorithmes, il faut être capable d'insérer un nouvel élément directement à sa place (ici *n* indique le numéro de la dernière case):

```
Algorithme Insertion (t : tableau d'entiers ; n,e : entier)
Variable i : entier
Début
  i ← n
  Tant que (i>0) et (t[i]>e) faire :
    t[i+1] ← t[i]
    i ← i-1
 Fin TQ
 t[i+1] ← e
Fin
```