Algorithmique et programmation : introduction

Algorithmique et programmation (Licence 1 - S2)

Introduction

Base fondamentale de l'informatique :

Comment faire faire des « actions compliquées » à un ordinateur à partir d'« actions simples » ?

- **Faire faire** : il s'agit d'une **description** des actions, pas des actions elles-mêmes ;
- **à un ordinateur** : l'ordinateur est *très* bête et *très* rigoureux : il faut être **précis**, **détaillé** et **intelligent**.

Algorithmique ou programmation

Définitions préliminaires :

- L'**algorithmique** est l'étude de la façon de décrire et décomposer des actions/calculs complexes en actions/calculs simples. Un **algorithme** est une telle décomposition.
- La **programmation** est l'étude de la traduction des algorithme dans un cadre compréhensible pour un **ordinateur**. Un **programme** est donc une traduction.

Notes historiques

- Premiers algorithmes connus : époque babylonienne (calculs commerciaux et fiscaux) ;
- « Algorithme » vient d'une déformation latine du nom du mathématicien musulman Muḥammad ibn Mūsā **al-Khwārizmī** (8ème siècle), du nom d'une province perse (actuellement plutôt en Ouzbékistan).
- Première machine programmable en 1801 (métier à tisser Jacquard), notion développée ensuite par Charles Babbage.

Algorithmes ?

Exemples d'algorithmes ?

1. Recette de cuisine.
2. Manuel de programmation d'un enregistrement sur lecteur DVD.
3. Itinéraire pour aller du bureau de la scolarité à la salle de TP micro 3.1.
4. Multiplication de nombres décimaux « à la main ».
5. ...

Quelques éléments d'algorithmiques

1. **Séquentialité** : *descendre l'escalier **puis** prendre à droite, **puis** tout droit jusqu'à la sortie, **puis** traverser la route, **puis**...*

   La séquentialité permet de décrire l'éxécution *consécutives* d'**instructions**.

2. **Sous-programmes** : *préparer 250 grammes de pâte brisée* ; *regarder le produit des deux chiffres sur la table de multiplication* ; ...

   Transformation d'un **bloc** d'instructions en **une** instruction (réutilisable). Note : usage de **paramètres**.

3. **Test** : ***si** vous voulez de la haute définition, choisissez un enregistrement sans perte.*

4. **Boucle** : *prendre la **4ème** porte à droite* ; *mélangez **jusqu'à** obtenir une préparation homogène* ; ***répéter** l'opération pour chaque chiffre du nombre à multiplier.*

   Répétition d'instructions beaucoup de fois, ou un nombre inconnu de fois.

Ce qui manque pour un algorithme

1. De la **rigueur** :
   - pas de justification aux actions → éxécution bête ;
   - aucune ambiguité possible ;
   - exécution reproductible.
2. Des **données** à manipuler : nombres (entiers, flottants), chaînes, ...
3. Un **langage** strict :
   - le français est naturellement ambigu...
   - ... et beaucoup trop complexe...
   - ... et beaucoup trop vaste.

Conception d'un algorithme : exemple

Exemple classique de problème à résoudre : le **tri**.

Commune	Population
Bordeaux	236725
Lille	226827
Lyon	479803
Marseille	850602
Montpellier	255080
Nantes	282047
Nice	340735
Paris	2234105
Reims	180842
Rennes	206604
Strasbourg	271708
Toulouse	440204

→

Commune	Population
	(1)
	(2)
	(3)
	(4)
	(5)
	(6)
	(7)
	(8)
	(9)
	(10)
	(11)
	(12)

Pourquoi le tri ?

Problème très classique en algorithmique :

1. utilisation fréquente ;
2. ne demande pas de données complexes (tableaux de nombres) ;
3. plusieurs solutions, des plus simples au plus complexes...
4. avec des avantages et des inconvénients.

Question pour la résolution du problème : **comment feriez-vous** ?

Rangement du maximum

Commune	Population
Bordeaux	236725
Lille	226827
Lyon	479803
Marseille	850602
Montpellier	255080
Nantes	282047
Nice	340735
~~**Paris**~~	~~**2234105**~~
Reims	180842
Rennes	206604
Strasbourg	271708
Toulouse	440204

→

Commune	Population
Paris	**2234105**
	(2)
	(3)
	(4)
	(5)
	(6)
	(7)
	(8)
	(9)
	(10)
	(11)
	(12)

Rangement du maximum (de ce qui reste)

Commune	Population
Bordeaux	236725
Lille	226827
Lyon	479803
~~Marseille~~	~~850602~~
Montpellier	255080
Nantes	282047
Nice	340735
~~Paris~~	~~2234105~~
Reims	180842
Rennes	206604
Strasbourg	271708
Toulouse	440204

→

Commune	Population
Paris	2234105
Marseille	**850602**
	(3)
	(4)
	(5)
	(6)
	(7)
	(8)
	(9)
	(10)
	(11)
	(12)

Etc...

Commune	Population
~~Bordeaux~~	~~236725~~
~~Lille~~	~~226827~~
~~Lyon~~	~~479803~~
~~Marseille~~	~~850602~~
~~Montpellier~~	~~255080~~
~~Nantes~~	~~282047~~
~~Nice~~	~~340735~~
~~Paris~~	~~2234105~~
~~Reims~~	~~180842~~
~~Rennes~~	~~206604~~
~~Strasbourg~~	~~271708~~
~~Toulouse~~	~~440204~~

→

Commune	Population
Paris	2234105
Marseille	850602
Lyon	479803
Toulouse	440204
Nice	340735
Nantes	282047
Strasbourg	271708
Montpellier	255080
Bordeaux	236725
Lille	226827
Rennes	206604
Reims	**180842**

Algorithme de tri

1. On choisit le plus grand (pour un tri par ordre décroissant) de l'ensemble à trier ;
2. on le range en haut du tableau et on l'enlève de l'ensemble à trier ;
3. si l'ensemble n'est pas vide, on reprend à l'étape 1.

Ce tri est appelé un **tri par sélection** (la première étape, la plus centrale dans ce tri, étant une « sélection » du plus grand élément).

Ecriture du tri par sélection

Pour écrire l'algorithme (ou le programmer), chacune des étapes doit être détaillées.

1. La première étape est elle-même un **problème algorithmique** (recherche du plus grand élément d'un ensemble). Elle demande encore une analyse (*comment feriez-vous ?*), peut-être d'autant plus difficile que cela semble simple. Cette décomposition d'un problème algorithmique en sous-problèmes est appelée **analyse descendante**.
2. Les étapes deux et trois sont des **opérations élémentaires** dépendantes du langage algorithmique (ou de programmation) utilisé. Savoir écrire ces étapes sans difficulté est avant tout une question de **pratique**, avec quelques choix à faire :
   - comment représenter les données (variables, tableaux, ...) ?
   - quelles structures du langage utiliser (boucles, fonctions, ...) ?

$\longrightarrow$ difficultés de deux ordres :

1. comment résoudre un problème (question « stratégique ») ;
2. comment écrire les étapes élémentaires (question « tactique »).

« Science » algorithmique

Avantage du tri par sélection :

1. utilise peu de déplacements de données.

Inconvénients du tri par sélection :

1. sur un ordinateur, les déplacements de données ne sont en général par un problème ;
2. utilise beaucoup de comparaisons → plutôt lent.

L'**algorithmique**, en tant que discipline, est la « science » (ou la branche de la science informatique) qui cherche des algorithmes pour résoudre des problèmes, et détermine leurs avantages et leurs inconvénients.

Langage algorithmique

Traditionnellement, on écrit les algorithmes dans un langage algorithmique (aussi appelé « pseudo-code »). C'est-à-dire un machin formalisé en simili-langage naturel. Par exemple :

```
lire a
affecter 0 à s
pour i allant de 1 à 10 faire
    affecter i+s à s
afficher s
```

Le langage algorithmique est essentiellement un langage de programmation sans certains détails techniques utilisés par l'ordinateur (par exemple, les déclarations de variables).

« Approches » de l'algorithmique

Il existe plusieurs approches dans la rédaction des algorithmes (et des programmes) : impérative, fonctionnelle, orientée objet, déclarative, ...

A savoir : toutes ces approches permettent de résoudre les mêmes problêmes. Aucun n'est plus « expressif » qu'un autre (par exemple, tous les langages permettent de programmer un tri par sélection). Les différences sont donc des questions de commodités.

Dans ce cours nous présenterons la forme « classique » de la programmation *impérative*.

Et les programmes ?

Langages de programmation : C, Basic, Java, Pascal, Caml, etc. et tous leurs dérivés... Plusieurs milliers de langages de programmation ont été créés.

Fonctionnalités différentes, approches différentes, capacités différentes...

Choix du **C** : langage impératif classique, encore largement utilisé, avec de nombreux dérivés, libre.

C

Langage apparu en **1972** (Dennis Ritchie, Bell Labs), inspiré de langages impératifs précédents (B, Algol, FORTRAN...). Initialement lié à un système d'exploitation, *Unix*.

Standardisations successives : 1978 (**C K&R**), 1989 (**ANSI C**), 1990 (**C ISO**), 1999 (**C99**).

Langage plutôt *bas-niveau* (proche du fonctionnement de la machine), ce qui le rend :

+ multi-usage, rapide, répandu, basique (utilisé pour créer les compilateurs et extensions d'autres langages) ;
− moyennement portable, peu flexible (dans un sens), peu rigoureux (donc demande de la rigueur de la part du programmeur), peu lisible.

Certains dérivés de C veulent corriger ces aspects (C++), d'autres langages s'en inspirent pour la *syntaxe* (Java).

Apprendre un langage

Un langage de programmation se définit par une **syntaxe** et une **sémantique**.

1. La **syntaxe** est l'ensemble des règles à règle à respecter pour écrire (et lire) un programme. Elle peut se comparer à la grammaire d'une langue naturelle. Exemple : *en C, les instructions se terminent par un point-virgule.*

   La syntaxe est vérifiée par l'ordinateur durant la **compilation** du programme.

2. La **sémantique** donne le « sens » du programme, c'est-à-dire ce que signifie les éléments du programme. Exemple : *en C, deux instructions successives sont éxecutées séquentiellement.*

   Un programme peut être syntaxiquement correct et ne pas faire ce qu'on veut (être « sémantiquement incorrect »), mais l'ordinateur ne peut pas le détecter.

Un programme en C

Un exemple de programme en C :

```
#include <stdio.h>
```

Lecture préalable d'un fichier nommé `stdio.h` : permet la *déclaration* de la fonction `puts`.

```
int main()
{
```

déclaration de la "fonction" `main` (ce qui est exécuté).

```
    puts("Mon premier programme.");
    return 0;
}
```

Instructions : les instructions **doivent** se terminer par un point-virgule.

le corps de la fonction est entre les accolades.

Compilation – édition de lien

Que faire du programme C (cas des TPs sous Linux) :

monprog.c

```
#include <stdio.h>


int main() {

   puts("Mon premier programme.");
    return 0;
}
```

Programme C

En ligne de commande :

```
gcc -Wall -o monprog monprog.c
```

Compilation

monprog.o

Fichier objet : décrit le contenu de la fonction `main`
Ne dit pas ce qu'est (ou ce que fait) `puts`.

Édition de liens

monprog

Fichier exécutable : peut être directement exécuté p
la machine.

Compilation + édition de lien

```
gcc -Wall -o monprog monprog.c
```

- `gcc` est le nom sous Linux du compilateur (au sens large) C.
- `monprog.c` est le nom du fichier contenant le programme (extension `.c` **obligatoire**).
- `-o monprog` indique le nom de l'exécutable à créer (option `-o` suivie du nom du programme).
- `-Wall` indique que le compilateur doit afficher tout ce qui lui semble « bizarre » sous forme d'avertissement (option **indispensable** en TP).

Si la compilation se passe sans problème, `gcc` n'affiche rien. Sinon, il est obligatoire de regarder les messages (avertissements ou erreurs) générés. Comprendre ces messages et (éventuellement) corriger leurs causes sont un objectif des TPs.

Exécution du programme

Le fichier `monprog` étant créé dans le répertoire courant, on tape `./monprog` suivi de la touche « entrée ». La réponse donne :

```
Mon premier programme.
```

Exemples d'erreurs/avertissements (1)

Programme :

```
int main() {
   puts("Mon premier programme.");
}
```

Résultat de `gcc -Wall` :

```
a.c: In function ‘main’:
a.c:2: attention : déclaration implicite de la fonction « puts »
a.c:3: attention : contrôle a atteint la fin non void de la fonction
```

Deux avertissements dans la fonction `main` :

1. Ligne 2 : il ne connaît pas la fonction `puts`. Cela signifie que la ligne `#include <stdio.h>` est manquante.
2. Ligne 3 : cet avertissement (un peu compliqué) n'a pas d'importance pour l'instant et peut être ignoré (cas unique). Il sera expliqué plus tard.

Exemples d'erreurs/avertissements (2)

Programme :

```
int main() {
   putf("Mon premier programme.");
}
```

Résultat de `gcc -Wall` :

```
a.c: In function ‘main’:
a.c:2: attention : déclaration implicite de la fonction « putf »
a.c:3: attention : contrôle a atteint la fin non void de la fonction
/tmp/cc6qdDma.o(.text+0xf): In function ‘main’:
: undefined reference to ‘putf’
collect2: ld a retourné 1 code d'état d'exécution
```

Deux avertissements et une erreur dans la fonction `main` :

1. Ligne 2 : il ne connaît pas la fonction `putf`. Cette fonction n'existe pas.
2. Ligne 3 : comme la dernière fois, on ignore cet avertissement.
3. L'erreur est écrite dans un code bizarre : en fait, c'est une erreur de l'édition de liens : il ne sait pas ce qu'est `putf`. A corriger donc.

Les avertissements n'empêchent pas la génération du programme mais indiquent que le code est douteux ou bizarre. Les erreurs bloquent la génération du programme.

Exemples d'erreurs/avertissements (3)

Autres exemples (erreurs de compilation) :

1. oubli du `;` à la fin de puts :

   ```
   a.c:5: erreur: erreur de syntaxe avant un élément lexical « } »
   ```

   Cette erreur est donnée sur la **ligne suivante**. Penser à vérifier le code qui précède.

2. oubli de l'accolade fermante `}` :

   ```
   a.c:4: erreur: erreur de syntaxe à la fin de l'entrée
   ```

   Erreur donnée sur un programme « incomplet ». Ce type d'oubli peut être beaucoup plus complexe à trouver, d'où l'intérêt de bien présenter son code.

3. mauvaise orthographe de `main` (par exemple `maan`) :

   ```
   /usr/lib/gcc/x86_64-redhat-linux/3.4.6/../../../../lib64/crt1.o(.text+0x21): In
   function `_start':
   : undefined reference to `main'
   collect2: ld a retourné 1 code d'état d'exécution
   ```

   Un programme C complet a besoin d'une fonction `main` pour fonctionner.

Objectifs du cours

1. Connaître les points principaux d'un **langage algorithmique**.
2. **Simuler** l'éxécution d'un algorithme (**dérouler** un algorithme).
3. **Écrire** soi-même un algorithme. Connaître certains algorithmes classiques (recherches, tris...).
4. Connaître les bases du langage **C**.
5. **Programmer** en C.

Site web

Cours sur http://intranet-depiup.univ-brest.fr/wiki/index.php/Algorithmique_et_programmation