Algorithmes et instructions

Temps d'étude conseillé : 9h.

1 Introduction aux algorithmes

1.1 Pseudo-langage

Vous connaissez ce terme depuis plusieurs années et vous l'avez tout autant manipulé. Donner une définition d'un algorithme est chose délicate car la définition est soit trop générale, soit trop longue car trop précise. Nous nous contenterons de la suivante.

Définition 1 (Algorithme). Suite d'instructions claires permettant de résoudre un problème.

On retrouve les algorithmes partout dans notre quotidien :

- dans une recette de cuisine,
- dans le calcul des racines d'un polynôme de degré 2,
- pour rechercher automatiquement une piste différentielle de probabilité maximale dans un réseau de substitution-permutation.

Nous pouvons remarquer que tous ces algorithmes possèdent quelques points communs. Pour commencer, ils ont tous pour objectif de répondre à un problème. Il est également classique pour un algorithme de posséder une ou plusieurs entrées / sorties. Et enfin, chaque algorithme détaille de façon précise les actions à mener pour répondre au problème.

Définition 2 (Composants d'un algorithme). Un algorithme possède obligatoirement les composants suivants :

- un titre clair et concis,
- une ou plusieurs entrées,
- une ou plusieurs sorties,
- des instructions claires détaillant l'algorithme.

La description d'un algorithme utilise un *pseudo-langage*. Il s'agit d'un langage (in)-formellement défini et adaptable à chaque utilisateur. L'idée est de trouver un vocabulaire commun que le plus grand nombre de personnes pourra comprendre. Pour cela, nous utiliserons le français et la liste de mots clés suivants, rédigés en majuscules :

- entrées, sorties : `Entrées`, `Sorties`, `Procédure` (ou `Algorithme`) ;
- instructions conditionnelles : `Si`, `Alors`, `Sinon`, `Instructions selon`, `Cas`, `Défaut` ;
- instructions de boucle : `Tant que`, `Faire`, `Pour`, `Pour chaque` ;
- instruction de fin : `Fin Si`, `Fin Tant que`, `Fin Pour`.

La première étape dans la rédaction d'un algorithme passe par la définition des entrées et sorties. Il faut pour cela étudier le contexte de l'algorithme, l'objectif et préciser les `Entrées` et `Sorties` de l'algorithme. Avant de rédiger le contenu de l'algorithme, vous devez rédiger l'ensemble des déclarations de variables en utilisant un procédé similaire à la déclaration des variables en C : définition du type suivi du nom de la variable, puis optionnellement l'affectation d'une valeur à l'aide du symbole ←.

La troisième étape consiste à rédiger l'algorithme en lui même, en manipulant les variables précédemment définies. Utilisez un brouillon afin de mettre au clair vos idées. Rédigez vos instructions du plus général au plus précis. Une fois l'algorithme complètement rédigé, il ne reste plus qu'à le tester sur quelques exemples afin de vérifier sa validité.

TD **Exercice 1.**

1.2 Traduction en langage C

Afin de terminer cette introduction, précisons quelques définitions permettant de traduire le pseudo-langage algorithmique en langage C.

Définition 3 (Expression). Une *expression* est une chaine de caractères formée d'opérateurs et d'opérandes.

Exemple 1. L'expression `a+2` correspond au calcul de la variable `a` auquel on ajoute la valeur 2. L'expression `a>=b` retourne 1 si `a` ≥ `b`, 0 sinon. L'expression `a=b` affecte la valeur de la variable `b` à la variable `a` et retourne la valeur de la variable `b`.

L'opérande peut donc être soit une variable, soit une valeur. L'opérateur est choisi parmi la liste suivante :

- opérateurs de calcul : `+` `-` `*` `/` `%` (modulo) ;
- opérateur d'assignation : `=` ;
- opérateurs de comparaison : `==` (test d'égalité) `!=` (test de différence) `>` `<` `>=` `<=` ;
- opérateurs logiques : `||` (ou) `&&` (et) `!` (non) ;
- manipulation binaires : `<<` `>>` `&` `|` `^` `~`

Définition 4 (Instruction). Une *instruction* est une chaine d'expressions terminée par un point virgule.

Un code C est donc formé d'un ensemble d'instructions. Certaines sont regroupées entre deux accolades. On appelle alors ces instructions ainsi regroupées un *bloc d'instructions*. Pour le moment, le seul bloc d'instructions que vous connaissez est l'ensemble des instructions que vous rédigez dans un `main()`.

Le but de ce chapitre est de détailler le moyen d'utiliser chacun des mots clés introduits ci-dessus. La même méthode sera utilisée à chaque fois. Dans un premier temps, vous devrez étudier un exemple. Puis vient la rédaction et l'implémentation de quelques algorithmes. Enfin, un exercice de synthèse vous sera proposé.

2 Instructions conditionnelles

2.1 Instruction `if/else`

Cette première instruction conditionnelle est la plus simple. En fonction du résultat d'un test logique, une série d'instructions est réalisée ou une autre. Il s'agit du classique algorithmique `Si/Sinon`.

TD **Exercice 2.**

Étudions maintenant la syntaxe d'une instruction conditionnelle en C. Le test `Si` se traduit par le mot clé `if`. L'expression à tester est alors à placer entre parenthèses juste après le mot clé `if`. Cette ligne **ne doit pas se terminer par un point-virgule**. Le `Sinon` se traduit par le mot clé `else`. Si l'expression retourne un résultat positif, alors on rentre dans le `if`. Si l'expression retourne un résultat nul, alors on rentre dans le `else`. Si plusieurs instructions dépendent d'un seul `if`, elles sont regroupées dans un bloc. Si une seule instruction dépend d'un `if`, les accolades ne sont pas utiles. On peut résumer cela à l'aide du parallèle algorithme / code C suivant.

Algorithme – Si-Sinon

```
Si test à valider alors
  Actions 1
Sinon
  Actions 2
Fin Si
```

```c
if (test)
{
  Actions 1;
}
else
{
  Actions 2;
}
```

Détaillons maintenant l'ensemble des opérateurs qu'il est possible d'utiliser au travers d'un `if`. Pour cela, on considère deux variables `a` et `b` de même type.

- `a==b` retourne 1 si $a = b$, 0 sinon.
- `a!=b` retourne 1 si $a \neq b$, 0 sinon.
- `a<b` retourne 1 si $a < b$, 0 sinon.
- `a<=b` retourne 1 si $a \leq b$, 0 sinon.
- `a>b` retourne 1 si $a > b$, 0 sinon.
- `a>=b` retourne 1 si $a \geq b$, 0 sinon.
- `a&&b` retourne 1 si $a \neq 0$ ET si $b \neq 0$, 0 sinon.
- `a||b` retourne 1 si $a \neq 0$ OU si $b \neq 0$, 0 sinon.

Exemple 2. Considérons les variables entières `a=3` et `b=-2`. On a alors :

- `a==b` retourne 0 ;
- `a!=b` retourne 1 ;
- `a>b` retourne 1 ;
- `(0<a)&&(a<5)` retourne 1 car $0 < a < 5$;
- `a||b` retourne 1.
- `a=b` retourne la valeur de `b`.

Remarque 1. Comme le montre le dernier exemple, l'expression `a=b` réalise l'affectation de la valeur `b` à `a` et retourne la valeur de `b`. La mettre dans un test n'a donc aucun sens ; n'oubliez pas que le test d'égalité est réalisé à l'aide de `==`. Attention également à l'expression `a&b` qui réalise un ET binaire entre les valeurs `a` et `b`, différent de `a&&b` qui réalise un ET logique entre les expressions `a` et `b`.

TD **Exercices 3 à 7.**

2.2 Instruction `switch/case`

Ce second type d'instruction doit vous être nouveau. Il est très proche du `Si/Sinon` mais se cantonne strictement au test d'égalité. Le `switch/case` est utilisé lorsque le codeur doit réaliser une série d'instructions dépendant de la valeur précise d'un test, en particulier si le test peut déboucher sur plusieurs résultats entiers différents.

TD **Exercice 8.**

L'exercice d'introduction précédent vous a permis de comprendre le fonctionnement global du `switch/case`. Résumons son fonctionnement, relativement proche du `if/else`. Le but est de tester la valeur d'une expression selon certains cas particuliers. L'expression à tester vient se placer entre les parenthèses du `switch`. Cette dernière doit avoir une valeur entière, sans quoi le code ne compilera pas. Des accolades viennent alors grouper les différents tests de l'expression inclus dans des `case`. La rédaction d'un cas commence par le mot clé `case:` suivi des instructions à réaliser. Le mot clé `break` permet de clore les instructions d'un cas sans exécuter celles du cas suivant. On résume alors le fonctionnement d'un `switch/case` de la façon suivante.

Algorithme 1 – Instructions selon

```
Instructions selon(exp)
    Cas val1: Instructions 1
    Cas val2: Instructions 2
    ...
    Défaut: Instructions D
Fin
```

```
switch (exp)  // Test sur exp
{
  case val1:  // Si exp == val1
    Instructions 1;
  break;      // optionnel

  case val2:  // Si exp == val2
    Instructions 2;
  .
  .
  .
  default:    // Par défaut
    Instructions D;
}
```

TD **Exercices 9 et 10.**

3 Instructions de boucle

3.1 Instructions `while, do/while`

Les boucles permettent de réaliser plusieurs fois d'affilée la même série d'instructions. La première instruction de boucle que nous allons découvrir est l'instruction `Tant Que` car sa syntaxe est particulièrement simple en C. Elle est utilisée lorsqu'une série d'instructions doit être réalisée tant qu'un test est vrai.

TD **Exercice 11.**

Comme vous venez de le remarquer, une boucle `while` permet de réaliser plusieurs fois les mêmes instructions. L'expression entre parenthèses représente le test à valider pour réaliser les instructions présentes dans le bloc. La boucle `do/while` est utilisée lorsqu'on veut exécuter au moins une fois les instructions de la boucle. Le test est donc placé après

```c
int croquettes, max;
printf("Combien de croquettes voulez-vous donner a Linux?\n");
scanf("%d", &croquettes);
max = croquettes;

while (croquettes > 0)
{
  printf("Linux mange une croquette.\n");
  croquettes--;
  printf("Il reste %d croquettes.\n", croquettes);
}

printf("Linux a fini de manger, mais il a encore faim!\n");
do
{
  printf("Linux vole une croquette de plus.\n");
  croquettes++;
  printf("En tout, %d croquettes bonus pour Linux!\n", croquettes);
}while (croquettes < max);
```

FIGURE 1 – Code permettant à Linux de se goinfrer.

la boucle. Remarquez dans ce cas que l'expression constituant le test se termine par un point-virgule. Dans les deux cas, l'incrément doit se faire dans les instructions de boucle, sans quoi une boucle infinie apparait. Résumons maintenant le fonctionnement du `while` et du `do/while`.

Algorithme 2 – `Tant que`

```
Tant que test faire
  Instructions
Fin Tant que
```

```c
while (test)
{
  Instructions;
}
```

Algorithme 3 – `Faire / Tant que`

```
Faire
  Instructions
Tant que test
```

```c
do
{
  Instructions;
}while (test);
```

TD **Exercices 12 à 14.**

3.2 Instruction `for`

L'instruction de boucle `Pour` est similaire au `Tant Que` dans le sens où elle permet de réaliser plusieurs fois une série d'instructions. L'avantage de cette instruction est qu'elle possède un compteur permettant de toujours savoir le nombre de tours de boucle réalisés et le nombre restant. C'est pour cela que tout bon codeur doit privilégier cette instruction au `Tant Que` lorsqu'il sait précisément à l'avance le nombre de tours de boucles à réaliser.

TD **Exercice 15.**

Une boucle for est constituée de quatre éléments : trois entre les parenthèses suivie de la série d'instructions à réaliser entre les accolades. Résumons l'utilité des expressions présentes dans les parenthèses et séparées par des points-virgule.

- La première expression est appelée *initialisation*. Elle regroupe l'ensemble des instructions à réaliser avant de dérouler la boucle. Habituellement, on y initialise le compteur de boucle à 0. Si plusieurs actions doivent être réalisées, on sépare chaque expression de la précédente par une virgule.
- La seconde expression est appelée *condition d'arrêt*. Votre professeur adoré préfère la nommer *condition de continuité* car la boucle s'exécute tant que cette expression est vrai. Ce que l'on inscrit ici peut être directement copié dans un `while` sans modifier le comportement du programme. Habituellement, on vérifie que le compteur n'est pas arrivé au bout du décompte.
- Enfin la troisième expression contient le ou les *incrément(s)*. On y place l'ensemble des actions à réaliser en fin de boucle, chacune étant séparé de la précédente par une virgule. Habituellement, on vient y incrémenter le compteur.

Il faut préciser que ces expressions sont facultatives. Aussi, la boucle `for(;;)` réalise une boucle infinie puisqu'elle ne contient pas de condition d'arrêt. On note enfin que le code C peut différer du langage algorithmique. En effet, le pseudo-langage privilégie la compréhension du code, alors que le langage C favorise l'optimisation. Ainsi, on peut résumer l'utilisation de la boucle `Pour` à l'aide de la comparaison suivante.

Algorithme 4 – `Boucle pour`

```
Pour compteur allant de début à fin faire
  Instructions
Fin Pour
```

```
for (initialisation; test; incrément)
{
  Instructions;
}
```

TD **Exercices 16 à 20.**

4 Trucs et astuces !

La Team Ibijau est heureuse de vous présenter sa première section « Trucs et astuces ». Au menu aujourd'hui, la génération de nombres aléatoire et la saisie fiabilisée. La Team Ibijau vous souhaite la plus grande réussite et vous dit à très bientôt.

4.1 Génération d'aléa

On souhaite souvent dans le programme que le hasard intervienne. Simuler le hasard est quelque chose de très compliqué. Pire, il est parfois capital que l'on puisse refaire exactement la même séquence de nombre aléatoire pour comparer l'efficacité de plusieurs algorithmes par exemple. On parle alors de séquences pseudo-aléatoires.

La fonction C de base dont vous avez besoin pour créer du hasard est la fonction `rand()` qui, à chaque appel, va renvoyer un nombre pseudo-aléatoire compris entre 0 et `RAND_MAX` (`man` vous dit qu'il faut inclure `stdlib.h`). Cette fonction part d'une situation de départ, appelée graine dans la littérature, pour générer la séquence. Dit autrement, à partir d'une même graine, vous verrez apparaître toujours la même séquence de nombres aléatoires. Le programme suivant par exemple génère 10 nombres aléatoirement entre 0 et `RAND_MAX`.

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

int main()
{
  int i;
  for (i=0; i<10; i++)
    printf("%d\n",rand());
  return 0;
}
```

Si vous souhaitez maintenant que le tirage soit compris entre 0 et 100 par exemple, la ligne contenant le `printf` est remplacée par :

```
printf("%d\n", (int)(100.*rand()/RAND_MAX));
```

Commentons un peu le code précédent : le code `rand()/RAND_MAX` permet de ramener les nombres tirés aléatoirement entre 0 et 1. Il suffit alors de multiplier le résultat précédent par 100 pour que le tirage soit alors compris entre 0 et 100. Mais à cette étape, le résultat est de type réel et on souhaiterait l'avoir en entier. Pour forcer la transformation de ce résultat en type entier, il faut mettre devant l'expression `(int)`. Cette opération s'appelle dans la littérature une conversion de type (cast en anglais) explicite.

Écrivez maintenant le programme précédent et exécutez-le plusieurs fois. Vous devez observez que vous obtenez toujours la même séquence de nombres, ce qui pour un tirage aléatoire n'est pas très satisfaisant. La raison en est que vous partez toujours de la même graine (i.e. situation de départ) lors de l'appel de `rand`.

Pour changer cela, il vous faut utiliser la fonction `srand` qui permet de fournir une nouvelle graine à la fonction `rand`. D'après `man`, le prototype de cette fonction est `void srand (unsigned int seed)`. La grande question est alors comment générer le nombre `seed` (graine en anglais) pour qu'il soit différent à chaque lancement du programme ?

Pour cela, on utilise classiquement l'horloge de la machine qui change en permanence via la fonction `time`. Cette fonction donne le nombre de secondes écoulées depuis le 1er janvier 1970 minuit GMT. Ainsi, pour peu qu'il s'écoule plus d'une seconde entre 2 exécutions successives, vous avez 2 graines différentes pour `srand` qui génèrera donc 2 situations de départ différentes pour `rand` et au final, vous obtiendrez 2 séquences aléatoires qui sembleront totalement décorrélées.

Pour fixer les idées, il suffit d'ajouter au code précédent l'instruction `srand(time(NULL))` en début de `main` pour obtenir à chaque lancement de l'exécutable une séquence de 10 nombres aléatoires différente de celle obtenue à l'exécution précédente.

TD Exercices 21 à 23.

4.2 scanf et les « retour charriot »

Copiez et exécutez le code de la figure 2.

```
int main()
{
  int i, res=0;
  while (res < 10)
  {
    i = scanf("%d", &res);
    printf ("i: %d\t-- %d --\n", i, res);
  }
  return 0;
}
```

FIGURE 2 – Utilisation risquée du `scanf`

Que se passe-t-il si vous utilisez des caractères alphabétiques ? Que proposez-vous pour contrer cela ? Sous Windows il est possible d'utiliser `fflush(stdin)`, malheureusement cela ne passe pas sous Linux sans effectuer une manipulation spéciale que vous ne verrez pas dans ce td. C'est pourquoi nous vous recommandons d'utiliser le code de la figure 3.

Le détail complet de ce code vous sera fourni dans un prochain TD (après avoir vu les fonctions). Mais vous pouvez d'ores et déjà l'utiliser dans vos codes.

5 Exercices d'approfondissement

TD Exercices 24 à 27.

```c
#include <stdio.h>

void clean_stdin(void)
{
  int c;
  do
  {
    c = getchar();
  } while (c != '\n' && c != EOF);
}

int main()
{
  int i, res=0;
  while (res < 10)
  {
    i = scanf("%d", &res);
    if (i<=0)
      clean_stdin();
    printf ("i: %d\t-- %d --\n", i, res);
  }
  return 0;
}
```

FIGURE 3 – Utilisation fiable du `scanf`