Introduction aux algorithmes

Algorithmique et structures de données, 2023-2024

P. Albuquerque (B410) et O. Malaspinas (A401), ISC, HEPIA

2023-09-19

En partie inspirés des supports de cours de P. Albuquerque

Qu'est-ce qu'un algorithme?

Définition informelle (recette)

- des entrées (les ingrédients, le matériel utilisé) ;
- des instructions élémentaires simples (frire, flamber, etc.), dont les exécutions dans un ordre précis amènent au résultat voulu ;
- un résultat : le plat préparé.

Qu'est-ce qu'un algorithme?

Définition informelle (recette)

- des entrées (les ingrédients, le matériel utilisé) ;
- des instructions élémentaires simples (frire, flamber, etc.), dont les exécutions dans un ordre précis amènent au résultat voulu ;
- un résultat : le plat préparé.

Histoire et étymologie

- Existent depuis 4500 ans au moins (algorithme de division, crible d'Eratosthène).
- Le mot algorithme est dérivé du nom du mathématicien perse *Muḥammad ibn Musā al-Khwārizmī*, qui a été "latinisé" comme *Algoritmi*.

Qu'est-ce qu'un algorithme?

Définition informelle (recette)

- des entrées (les ingrédients, le matériel utilisé) ;
- des instructions élémentaires simples (frire, flamber, etc.), dont les exécutions dans un ordre précis amènent au résultat voulu ;
- un résultat : le plat préparé.

Histoire et étymologie

- Existent depuis 4500 ans au moins (algorithme de division, crible d'Eratosthène).
- Le mot algorithme est dérivé du nom du mathématicien perse *Muḥammad ibn Musā al-Khwārizmī*, qui a été "latinisé" comme *Algoritmi*.

Définition formelle

En partant d'un état initial et d'entrées (peut-être vides), une séquence finie d'instruction bien définies (ordonnées) implémentables sur un ordinateur, afin de résoudre typiquement une classe de problèmes ou effectuer un calcul.

Notions de base d'algorithmique

Variable

Notions de base d'algorithmique

Variable

- Paire: identifiant - valeur (assignation);

Séquence d'instructions / expressions

Notions de base d'algorithmique

Variable

- Paire: identifiant - valeur (assignation);

Séquence d'instructions / expressions

- Opérateurs (arithmétiques / booléens)
- Boucles;
- Structures de contrôle;
- Fonctions;

Algorithme de vérification qu'un nombre est premier (1/3)

Nombre premier: nombre possédant deux diviseurs entiers distincts.

Algorithme de vérification qu'un nombre est premier (1/3)

Nombre premier: nombre possédant deux diviseurs entiers distincts.

Algorithme naïf (problème)

```
booléen est_premier(nombre)
    si
        pour tout i, t.q. 1 < i < nombre
            i ne divise pas nombre
    alors vrai
    sinon faux
```

Algorithme de vérification qu'un nombre est premier (1/3)

Nombre premier: nombre possédant deux diviseurs entiers distincts.

Algorithme naïf (problème)

```
booléen est_premier(nombre)
    si
        pour tout i, t.q. 1 < i < nombre
            i ne divise pas nombre
    alors vrai
    sinon faux
```

Pas vraiment un algorithme: pas une séquence ordonnée et bien définie

Algorithme de vérification qu'un nombre est premier (1/3)

Nombre premier: nombre possédant deux diviseurs entiers distincts.

Algorithme naïf (problème)

```
booléen est_premier(nombre)
    si
        pour tout i, t.q. 1 < i < nombre
            i ne divise pas nombre
    alors vrai
    sinon faux
```

Pas vraiment un algorithme: pas une séquence ordonnée et bien définie

Problème: Comment écrire ça sous une forme algorithmique?

Algorithme de vérification qu'un nombre est premier (2/3)

Algorithme naïf (une solution)

```
booléen est_premier(nombre) // fonction
    soit i = 2;         // variable, type, assignation
    tant que i < nombre // boucle
        si nombre modulo i == 0 // expression typée
            retourne faux    // expression typée
        i = i + 1
    retourne vrai // expression typée
```

Algorithme de vérification qu'un nombre est premier (3/3)

Algorithme naïf (une solution en C)

```c
bool est_premier(int nombre) {
    int i; // i est un entier
    i = 2; // assignation i à 2
    while (i < nombre) { // boucle avec condition
        if (0 == nombre % i) { // is i divise nombre
            return false; // i n'est pas premier
        }
        i = i + 1; // sinon on incrémente i
    }
    return true;
}
```

Algorithme de vérification qu'un nombre est premier (3/3)

Algorithme naïf (une solution en C)

```c
bool est_premier(int nombre) {
    int i; // i est un entier
    i = 2; // assignation i à 2
    while (i < nombre) { // boucle avec condition
        if (0 == nombre % i) { // is i divise nombre
            return false; // i n'est pas premier
        }
        i = i + 1; // sinon on incrémente i
    }
    return true;
}
```

Exercice: Comment faire plus rapide?

Génération d'un exécutable

- Pour pouvoir être exécuté un code C doit être d'abord compilé (avec `gcc` ou `clang`).
- Pour un code `prog.c` la compilation "minimale" est

```
$ gcc prog.c
$ ./a.out # exécutable par défaut
```

- Il existe une multitude d'options de compilation:

```
$ gcc -O1 -std=c11 -Wall -Wextra -g porg.c -o prog
    -fsanitize=address
```

1. `-std=c11` utilisation de C11.
2. `-Wall et -Wextra` activation des warnings.
3. `-fsanitize=...` contrôles d'erreurs à l'exécution (coût en performance).
4. `-g` symboles de débogages sont gardés.
5. `-o` défini le fichier exécutable à produire en sortie.
6. `-O1`, `-O2`, `-O3`: activation de divers degrés d'optimisation

La simplicité de C?

32 mots-clé et c'est tout

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Déclaration et typage

En C lorsqu'on veut utiliser une variable (ou une constante), on doit déclarer son type

```c
const double two = 2.0; // déclaration et init.
int x;   // déclaration (instruction)
char c;  // déclaration (instruction)
x = 1;   // affectation (expression)
c = 'a'; // affectation (expression)
int y = x; // déclaration et initialisation en même temps
int a, b, c; // déclarations multiples
a = b = c = 1; // init. multiples
```

Les variables (1/2)

Variables et portée

- Une variable est un identifiant, qui peut être liée à une valeur (un expression).
- Une variable a une **portée** qui définit où elle est *visible* (où elle peut être accédée).
- La portée est **globale** ou **locale**.
- Une variable est **globale** est accessible à tout endroit d'un programme et doit être déclarée en dehors de toute fonction.
- Une variable est **locale** lorsqu'elle est déclarée dans un **bloc**, `{...}`.
- Une variable est dans la portée **après** avoir été déclarée.

Les variables (2/2)

Exemple

```
float max; // variable globale accessible partout
int foo() {
    // max est visible ici
    float a = max; // valide
    // par contre les varibles du main() ne sont pas visibles
}
int main() {
    // max est visible ici
    int x = 1; // x est locale à main
    {
        // x est visible ici, y pas encore
        // on peut par exemple pas faire x = y;
        int y = 2;
    } // y est détruite à la sortie du bloc
} // x est à la sortie de main
```

Quiz: compile ou compile pas?

Quiz: compile ou compile pas

Types de base (1/4)

Numériques

Type	Signification (**gcc pour x86-64**)
`char`, `unsigned char`	Entier signé/non-signé 8-bit
`short`, `unsigned short`	Entier signé/non-signé 16-bit
`int`, `unsigned int`	Entier signé/non-signé 32-bit
`long`, `unsigned long`	Entier signé/non-signé 64-bit
`float`	Nombre à virgule flottante, simple précision
`double`	Nombre à virgule flottante, double précision

La signification de `short`, `int`, ... dépend du compilateur et de l'architecture.

Types de base (2/4)

Voir `<stdint.h>` pour des représentations **portables**

Type	Signification
`int8_t`, `uint8_t`	Entier signé/non-signé 8-bit
`int16_t`, `uint16_t`	Entier signé/non-signé 16-bit
`int32_t`, `uint32_t`	Entier signé/non-signé 32-bit
`int64_t`, `uint64_t`	Entier signé/non-signé 64-bit

Types de base (2/4)

Voir `<stdint.h>` pour des représentations **portables**

Type	Signification
`int8_t`, `uint8_t`	Entier signé/non-signé 8-bit
`int16_t`, `uint16_t`	Entier signé/non-signé 16-bit
`int32_t`, `uint32_t`	Entier signé/non-signé 32-bit
`int64_t`, `uint64_t`	Entier signé/non-signé 64-bit

Prenez l'habitude d'utiliser ces types-là!

Types de base (3/4)

Booléens

- Le ANSI C n'offre pas de booléens.
- L'entier `0` signifie *faux*, tout le reste *vrai*.
- Depuis C99, la librairie `stdbool` met à disposition un type `bool`.
- En réalité c'est un entier:
  - $1 \Rightarrow$ `true`
  - $0 \Rightarrow$ `false`
- On peut les manipuler comme des entier (les sommer, les multiplier, ...).

Quiz: booléens

Quiz: booléens

Types de base (4/4)

Conversions

- Les conversions se font de manière:
  - Explicite:

```
int a = (int)2.8;
double b = (double)a;
int c = (int)(2.8+0.5);
```

  - Implicite:

```
int a = 2.8; // warning, si activés, avec clang
double b = a + 0.5;
char c = b; // pas de warning...
int d = 'c';
```

Quiz: conversions

Quiz: conversions

Expressions et opérateurs (1/6)

Une expression est tout bout de code qui est **évalué**.

Expressions simples

- Pas d'opérateurs impliqués.
- Les littéraux, les variables, et les constantes.

```
const int L = -1; // 'L' est une constante, -1 un littéral
int x = 0;        // '0' est un litéral
int y = x;        // 'x' est une variable
int z = L;        // 'L' est une constante
```

Expressions complexes

- Obtenues en combinant des *opérandes* avec des *opérateurs*

```
int x;     // pas une expression (une instruction)
x = 4 + 5; // 4 + 5 est une expression
           // dont le résultat est affecté à 'x'
```

Expressions et opérateurs (2/6)

Opérateurs relationnels

Opérateurs testant la relation entre deux *expressions*:

- `(a opérateur b)` retourne `1` si l'expression s'évalue à `true`, `0` si l'expression s'évalue à `false`.

Opérateur	Syntaxe	Résultat
`<`	`a < b`	1 si a $<$ b; 0 sinon
`>`	`a > b`	1 si a $>$ b; 0 sinon
`<=`	`a <= b`	1 si a $<=$ b; 0 sinon
`>=`	`a >= b`	1 si a $>=$ b; 0 sinon
`==`	`a == b`	1 si a $==$ b; 0 sinon
`!=`	`a != b`	1 si a != b; 0 sinon

Expressions et opérateurs (3/6)

Opérateurs logiques

Opérateur	Syntaxe	Signification
`&&`	`a && b`	ET logique
`||`	`a || b`	OU logique
`!`	`!a`	NON logique

Quiz: opérateurs logiques

Expressions et opérateurs (4/6)

Opérateurs arithmétiques

Opérateur	Syntaxe	Signification
`+`	`a + b`	Addition
`-`	`a - b`	Soustraction
`*`	`a * b`	Multiplication
`/`	`a / b`	Division
`%`	`a % b`	Modulo

Expressions et opérateurs (5/6)

Opérateurs d'assignation

Opérateur	Syntaxe	Signification
`=`	`a = b`	Affecte la valeur `b` à la variable `a` et retourne la valeur de `b`
`+=`	`a += b`	Additionne la valeur de `b` à `a` et assigne le résultat à `a`.
`-=`	`a -= b`	Soustrait la valeur de `b` à `a` et assigne le résultat à `a`.
`*=`	`a *= b`	Multiplie la valeur de `b` à `a` et assigne le résultat à `a`.
`/=`	`a /= b`	Divise la valeur de `b` à `a` et assigne le résultat à `a`.
`%=`	`a %= b`	Calcule le modulo la valeur de `b` à `a` et assigne le résultat à `a`.

Expressions et opérateurs (6/6)

Opérateurs d'assignation (suite)

Opérateur	Syntaxe	Signification
`++`	`++a`	Incrémente la valeur de a de 1 et retourne le résultat (`a += 1`).
`--`	`--a`	Décrémente la valeur de a de 1 et retourne le résultat (`a -= 1`).
`++`	`a++`	Retourne a et incrémente a de 1.
`--`	`a--`	Retourne a et décrémente a de 1.

Structures de contrôle: if .. else if .. else (1/2)

Syntaxe

```
if (expression) {
    instructions;
} else if (expression) { // optionnel
                         // il peut y en avoir plusieurs
    instructions;
} else {
    instructions; // optionnel
}
```

```
if (x) { // si x s'évalue à `vrai`
    printf("x s'évalue à vrai.\n");
} else if (y == 8) { // si y vaut 8
    printf("y vaut 8.\n");
} else {
    printf("Ni l'un ni l'autre.\n");
}
```

Structures de contrôle: if .. else if .. else (2/2)

Pièges

```
int x, y;
x = y = 3;
if (x = 2)
    printf("x = 2 est vrai.\n");
else if (y < 8)
    printf("y < 8.\n");
else if (y == 3)
    printf("y vaut 3 mais cela ne sera jamais affiché.\n");
else
    printf("Ni l'un ni l'autre.\n");
    x = -1; // toujours évalué
```

Quiz: `if ... else`

Quiz: `if ... else`

Structures de contrôle: while

La boucle while

```
while (condition) {
    instructions;
}
do {
    instructions;
} while (condition);
```

La boucle while, un exemple

```
int sum = 0; // syntaxe C99
while (sum < 10) {
    sum += 1;
}
do {
    sum += 10;
} while (sum < 100)
```

Structures de contrôle: for

La boucle for

```
for (expression1; expression2; expression3) {
    instructions;
}
```

La boucle for

```
int sum = 0; // syntaxe C99
for (int i = 0; i < 10; i++) {
    sum += i;
}

for (int i = 0; i != 1; i = rand() % 4) { // ésotérique
    printf("C'est plus ésotérique.\n");
}
```