Structures de données et algorithmes

Yann Strozecki
yann.strozecki@uvsq.fr

Septembre 2016

Organisation du cours

- 12 séances de cours et un examen
- deux interros en cours et un contrôle à la séance 7
- 2 groupes de TD le lundi matin et le mardi matin
- un devoir à la maison
- poly de td et autres ressources sur e-campus 2

Introduction

Les 3 niveaux d'abstraction :

- **Les problèmes** décrits en langage naturel.
- **Les algorithmes** décrits dans un pseudo-langage de programmation, proche du langage naturel.
- **Les programmes** décrits dans un langage de programmation (C, Caml, Pascal, C++, Java, PHP, Python ...).
- **Le code machine** est une séquence d'instructions du processeur.
- **Un stockage magnétique ou électrique** qui est modifié selon les lois de la physique.

Algorithmes et programmes sont des versions concises de ce qui va se passer sur l'ordinateur.

Introduction

Les 3 niveaux d'abstraction :

- **Les problèmes** décrits en langage naturel.
- **Les algorithmes** décrits dans un pseudo-langage de programmation, proche du langage naturel.
- **Les programmes** décrits dans un langage de programmation (C, Caml, Pascal, C++, Java, PHP, Python ...).
- **Le code machine** est une séquence d'instructions du processeur.
- **Un stockage magnétique ou électrique** qui est modifié selon les lois de la physique.

Algorithmes et programmes sont des versions concises de ce qui va se passer sur l'ordinateur.

Démarche de résolution d'un problème

Conceptuel

- Comment le résoudre ?
- Peut-on utiliser des méthodes classiques ?
- Comment représenter et organiser les données ?

Résolution du problème => Algorithme

Technique

- Comment mettre en œuvre mon algorithme sur une machine ?
- Quelles sont les ressources à ma disposition ?
- Quel est le langage le plus adapté ?

Qu'est ce qu'un algorithme ?

Quelques exemples :

- Une recette de cuisine
- Résolution d'une équation du second degré (al-Khwarizmi).
- Stratégie pour gagner au morpion
- Arbre de décision et système expert

Définitions :

- An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function (Wikipedia).
- A computer is a storyteller and algorithms are its tales (Chazelle).
- An algorithm is any of my machines (Turing).
- An agorithm is a recursive function (Church).

Qu'est ce qu'un algorithme ?

Quelques exemples :

- Une recette de cuisine
- Résolution d'une équation du second degré (al-Khwarizmi).
- Stratégie pour gagner au morpion
- Arbre de décision et système expert

Définitions :

- An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function (Wikipedia).
- A computer is a storyteller and algorithms are its tales (Chazelle).
- An algorithm is any of my machines (Turing).
- An agorithm is a recursive function (Church).

Questions relatives aux algorithmes

Questions à se poser une fois l'algorithme écrit :

- Les sorties correspondent-elles à la solution de mon problème ?
  Preuve (correction) de l'algorithme

- Est-ce que l'algorithme termine ?
  Terminaison de l'algorithme

Questions relatives aux algorithmes

Questions à se poser une fois l'algorithme écrit :

- Les sorties correspondent-elles à la solution de mon problème ?
  Preuve (correction) de l'algorithme
- Est-ce que l'algorithme termine ?
  Terminaison de l'algorithme
- Combien de calculs élémentaires doit-on faire pour produire la sortie ?
  Complexité en temps de l'algorithme

Questions relatives aux algorithmes

Questions à se poser une fois l'algorithme écrit :

- Les sorties correspondent-elles à la solution de mon problème ?
  Preuve (correction) de l'algorithme
- Est-ce que l'algorithme termine ?
  Terminaison de l'algorithme
- Combien de calculs élémentaires doit-on faire pour produire la sortie ?
  Complexité en temps de l'algorithme
- Combien de mémoire utilise l'algorithme en cours de fonctionnement ?
  Complexité en espace de l'algorithme

Questions relatives aux algorithmes

Questions à se poser une fois l'algorithme écrit :

- Les sorties correspondent-elles à la solution de mon problème ?
  Preuve (correction) de l'algorithme
- Est-ce que l'algorithme termine ?
  Terminaison de l'algorithme
- Combien de calculs élémentaires doit-on faire pour produire la sortie ?
  Complexité en temps de l'algorithme
- Combien de mémoire utilise l'algorithme en cours de fonctionnement ?
  Complexité en espace de l'algorithme

Les références utiles :

- The Witness
- Human Ressource Machine
- Braid
- Portal 1 et 2
- World of Goo
- The Thalos Principle

Les autres références utiles :

- Algorithmes. Cormen.
- Introduction à l'algorithmique de Cormen, Leiserson et Rivest.
- Algorithm design. Kleinberg et Tardos.
- The algorithm design manual. Skiena.
- Exercices et problèmes d'algorithmiques de Baynat, Chrétienne, Hanen.
- Les pages wikipedia des thèmes abordés.

Le pseudo-code

En algorithmique on est moins précis qu'en programmation et surtout, on utilise le langage le plus simple possible pour faciliter l'analyse. La brique de base d'un programme s'appelle un terme :

- ils ont un type : entiers, flottants, booléens ...
- on peut faire des tableaux d'objets des types précédents indicés à partir de 0

Le pseudo-code

En algorithmique on est moins précis qu'en programmation et surtout, on utilise le langage le plus simple possible pour faciliter l'analyse. La brique de base d'un programme s'appelle un terme :

- ils ont un type : entiers, flottants, booléens ...
- on peut faire des tableaux d'objets des types précédents indicés à partir de 0
- les variables (typées) sont des termes

Le pseudo-code

En algorithmique on est moins précis qu'en programmation et surtout, on utilise le langage le plus simple possible pour faciliter l'analyse. La brique de base d'un programme s'appelle un terme :

- ils ont un type : entiers, flottants, booléens ...
- on peut faire des tableaux d'objets des types précédents indicés à partir de 0
- les variables (typées) sont des termes
- on peut appliquer des opérations à deux termes de même type pour obtenir un nouveau terme, addition $+$, multiplication $*$, comparaisons $<$ et $==$ ...

Le pseudo-code

En algorithmique on est moins précis qu'en programmation et surtout, on utilise le langage le plus simple possible pour faciliter l'analyse. La brique de base d'un programme s'appelle un terme :

- ils ont un type : entiers, flottants, booléens ...
- on peut faire des tableaux d'objets des types précédents indicés à partir de 0
- les variables (typées) sont des termes
- on peut appliquer des opérations à deux termes de même type pour obtenir un nouveau terme, addition $+$, multiplication $*$, comparaisons $<$ et $==$ ...

Il est bon de préciser les entrées et sorties d'un algorithme ainsi que leur type.

Le pseudo-code

En algorithmique on est moins précis qu'en programmation et surtout, on utilise le langage le plus simple possible pour faciliter l'analyse. La brique de base d'un programme s'appelle un terme :

- ils ont un type : entiers, flottants, booléens ...
- on peut faire des tableaux d'objets des types précédents indicés à partir de 0
- les variables (typées) sont des termes
- on peut appliquer des opérations à deux termes de même type pour obtenir un nouveau terme, addition $+$, multiplication $*$, comparaisons $<$ et $==$ ...

Il est bon de préciser les entrées et sorties d'un algorithme ainsi que leur type.

C'est l'histoire d'un type

Un type est un ensemble d'objets. Quand on donne un type à une variable on restreint les valeurs qu'elle peut prendre.

Par exemple, qu'est ce qui est correct avec le type Légume ?

Légume x = courgette

Légume y = haricot

Légume z = fraise

rutabaga + 3

C'est l'histoire d'un type

Un type est un ensemble d'objets. Quand on donne un type à une variable on restreint les valeurs qu'elle peut prendre.

Par exemple, qu'est ce qui est correct avec le type Légume ?
Légume x = courgette
Légume y = haricot
Légume z = fraise
rutabaga + 3

Donner un type c'est comme commenter, ça aide à comprendre le code et ça évite des erreurs.

C'est l'histoire d'un type

Un type est un ensemble d'objets. Quand on donne un type à une variable on restreint les valeurs qu'elle peut prendre.

Par exemple, qu'est ce qui est correct avec le type Légume ?
Légume x = courgette
Légume y = haricot
Légume z = fraise
rutabaga + 3

Donner un type c'est comme commenter, ça aide à comprendre le code et ça évite des erreurs.

En pseudo code, on peut créer des types complexes en définissant des structures, des tableaux ou des références.

C'est l'histoire d'un type

Un type est un ensemble d'objets. Quand on donne un type à une variable on restreint les valeurs qu'elle peut prendre.

Par exemple, qu'est ce qui est correct avec le type Légume ?
Légume x = courgette
Légume y = haricot
Légume z = fraise
rutabaga + 3

Donner un type c'est comme commenter, ça aide à comprendre le code et ça évite des erreurs.

En pseudo code, on peut créer des types complexes en définissant des structures, des tableaux ou des références.

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite
- retour d'un valeur

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite
- retour d'un valeur
- appel de fonctions déjà décrites avec passage par **valeur**

On peut ensuite composer ces instructions en les enchainant et les répétants :

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite
- retour d'un valeur
- appel de fonctions déjà décrites avec passage par **valeur**

On peut ensuite composer ces instructions en les enchainant et les répétants :

- composition dénotée par un saut de ligne

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite
- retour d'un valeur
- appel de fonctions déjà décrites avec passage par **valeur**

On peut ensuite composer ces instructions en les enchainant et les répétants :

- composition dénotée par un saut de ligne
- if(test) then(action1) else(action2)

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite
- retour d'un valeur
- appel de fonctions déjà décrites avec passage par **valeur**

On peut ensuite composer ces instructions en les enchainant et les répétants :

- composition dénotée par un saut de ligne
- if(test) then(action1) else(action2)
- boucle : while(condition) faire(action)

Le pseudo-code (suite)

A partir des termes on peut écrire des instructions, qui servent à modifier la mémoire. En C, elles sont terminées par un point virgule.

- déclaration de variable
- affectation $\leftarrow$ **de la droite vers la gauche** avec une variable à gauche et une valeur à droite
- retour d'un valeur
- appel de fonctions déjà décrites avec passage par **valeur**

On peut ensuite composer ces instructions en les enchainant et les répétants :

- composition dénotée par un saut de ligne
- if(test) then(action1) else(action2)
- boucle : while(condition) faire(action)

Un exemple simple d'algorithme

Algorithme 1 : Arithmétique

Data : n : entier

```
1 while n ≠ 0 do
2     if n  mod 2 = 0 then
3         n ← n/2
4     else
5         n ← n + 1
```

Ce programme termine-t-il ?

Que se passe-t-il si on divise par 3 au lieu de 2 ?

Efficacité d'un algorithme

- Mesure intrinsèque de la complexité de l'algorithme indépendamment de l'implémentation.
- Ne dépend pas de l'ordinateur sur lequel est exécuté le programme.
- Compte le nombre d'opérations élémentaires d'un algorithme par exemple une addition ou une écriture en mémoire.
- C'est une mesure approximative mais suffisante pour permettre la comparaison entre différents algorithmes pour un même problème.

Exemple de complexité

Recherche de l'algorithme avec le moins d'étapes élémentaires.

Somme des nombres de 1 à n

Idée 1 : Utilisation d'une boucle

Algorithme 1

$i \leftarrow 1$

$som \leftarrow 0$

Tant que $i \leq n$ *Faire*

$som \leftarrow som + i$

$i \leftarrow i + 1$

Fin Tant que

Coût : $2 \times n$ additions

Exemple de complexité (2)

Recherche de l'algorithme avec le moins d'étapes élémentaires.

Somme des nombres de 1 à n

Idée 2 : Utilisation des mathématiques

$$\sum_{i=1}^{n} i = \frac{n \times (n+1)}{2}$$

Algorithme 2

$som \leftarrow n + 1$

$som \leftarrow som * n$

$som \leftarrow som/2$

Coût : 1 addition, 1 multiplication et 1 division.

Précisions sur la complexité

- Mesure élémentaire :
  - nombre de comparaisons
  - nombre d'affectations
  - nombre d'opérations arithmétiques
  - ...

- On cherche la complexité d'un algorithme $\mathcal{A}$ en fonction de la taille des entrées.

- La complexité ne peut pas être calculée automatiquement par l'ordinateur. Par contre on peut faire du *profiling*.

Quelques règles

$cout(x)$: nbre d'op. élémentaires de l'ens. d'instructions x.

Séquence d'instructions : $x_1; x_2;$

$$cout(x_1; x_2;) = cout(x_1;) + cout(x_2;)$$

Exemple

Mesure : nombre d'opérations arithmétiques.

Algorithme $\mathcal{A}$
Début
$som \leftarrow n + 1$
$som \leftarrow som * n$
$som \leftarrow som/2$
Fin

$$cout(\mathcal{A}) = cout(som \leftarrow n + 1) + cout(som \leftarrow som * n) + cout(som \leftarrow som/2) = 3$$

Quelques règles (2)

Les boucles simples : *tant que* $condition$ *faire* x_i ;

$$cout(boucle) = \sum_{i=1}^{n}(cout(x_i) + cout(condition))$$

Exemple

Mesure : nombre de comparaisons.

Algorithme $\mathcal{B}$
Début
❶ $i \leftarrow 1$
❷ $som \leftarrow 0$
Tant que $i \leq n$ *Faire*
❸ $som \leftarrow som + i$
❹ $i \leftarrow i + 1$
Fin Tant que
Fin

$$cout(\mathcal{B}) = cout(❶;❷) + \sum_{i=1}^{n}(cout(i \leq n) + cout(❸;❹)) = n$$

Quelques règles (3)

Conditionnelle : *Si* $condition$ *alors* x_{vrai}; *sinon* x_{faux};

$$cout(conditionnelle) \leq cout(condition) + \max(cout(x_{vrai}); cout(x_{faux}))$$

Exemple

Mesure : nombre d'affectations.

Algorithme $\mathcal{C}$
Début
❶ $u \leftarrow 0$
Si $i \mod 2 = 0$ *Alors*
❷ $u \leftarrow i/2$
Sinon
❸ $u \leftarrow i - 1$
❹ $u \leftarrow u/2$
Fin Si
Fin

$$cout(\mathcal{C}) \leq cout(❶) + cout(i \mod 2 = 0) + \max(cout(❷); cout(❸;❹)) = 3$$

Quelques règles (4)

Appel de fonction : $fonction(x);$

Exemple

Mesure : nombre d'affectations.

Algorithme $\mathcal{D}$
Début
❶ $u \leftarrow 0$
❷ $i \leftarrow 0$
Tant que $i < n$
❸ $u \leftarrow u + \mathcal{B}(i)$
❹ $i \leftarrow i + 1$
Fin Tant que
Fin

$$cout(\mathcal{D}) = cout(❶;❷) + \sum_{i=1}^{n}(cout(i \leq n) + 2 + cout(\mathcal{B}(i))) = 2n^2 + 4n + 2$$

Un exemple : produit de deux matrices carrées $C = A \times B$

Deux matrices $A = (a_{ij})$ et $B = (b_{ij})$ de dimensions $n \times n$.

$$\forall i, j,\ c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Cette formule mathématique se traduit *littéralement* en programme au tableau.

- **Opération élémentaire :** la multiplication.
- **Coût de l'algorithme :**

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} cout(C[i][j] \leftarrow \dots) = n^3$$

Un exemple : produit de deux matrices carrées $C = A \times B$

Deux matrices $A = (a_{ij})$ et $B = (b_{ij})$ de dimensions $n \times n$.

$$\forall i, j,\ c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Cette formule mathématique se traduit *littéralement* en programme au tableau.

- **Opération élémentaire :** la multiplication.
- **Coût de l'algorithme :**

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} cout(C[i][j] \leftarrow \dots) = n^3$$

- Il existe d'autres algorithmes plus performants !

Un exemple : produit de deux matrices carrées $C = A \times B$

Deux matrices $A = (a_{ij})$ et $B = (b_{ij})$ de dimensions $n \times n$.

$$\forall i, j,\ c_{ij} = \sum_{k=1}^{n} a_{ik} \times b_{kj}$$

Cette formule mathématique se traduit *littéralement* en programme au tableau.

- **Opération élémentaire :** la multiplication.
- **Coût de l'algorithme :**

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} cout(C[i][j] \leftarrow \dots) = n^3$$

- Il existe d'autres algorithmes plus performants !

Analyse de la complexité d'un programme quelconque

Algorithme inconnu

$i \leftarrow 1$
$y \leftarrow 0$
Tant que $i \leq n$ *Faire*
    $i \leftarrow i + 1$
    $j \leftarrow 1$
    Tant que $j \leq x$ *Faire*
        $y \leftarrow y + i \mod j$
    Fin Tant que
Fin Tant que

À vous de trouver le coût précis de cet algorithme.

Analyse de la complexité d'un programme quelconque

Algorithme inconnu

$i \leftarrow 1$
$y \leftarrow 0$
Tant que $i \leq n$ *Faire*
    $i \leftarrow i + 1$
    $j \leftarrow 1$
    Tant que $j \leq x$ *Faire*
        $y \leftarrow y + i \mod j$
        $j \leftarrow j + 1$
    Fin Tant que
Fin Tant que

À vous de trouver le coût précis de cet algorithme.

Analyse de la complexité d'un programme quelconque

Algorithme inconnu

$i \leftarrow 1$
$y \leftarrow 0$
Tant que $i \leq n$ *Faire*
    $i \leftarrow i + 1$
    $j \leftarrow 1$
    Tant que $j \leq x$ *Faire*
        $y \leftarrow y + i \mod j$
        $j \leftarrow j + 1$
    Fin Tant que
Fin Tant que

Coût : environ n^2

À vous de trouver le coût précis de cet algorithme.

L'âge du capitaine

Écrire un algorithme de 8 lignes au plus utilisant uniquement comme opération de base des sommes, des produits et des comparaisons d'entiers. Vous pouvez utiliser au plus 3 variables x,y,z.

Objectif : si je donne des valeurs initiales de x,y et z par exemple 2, 3 et 4 votre voisin doit être incapable de trouver la valeur de x à la fin du programme.

Objectif avancé : je dois aussi être incapable de trouver la valeur de x.

L'âge du capitaine

Écrire un algorithme de 8 lignes au plus utilisant uniquement comme opération de base des sommes, des produits et des comparaisons d'entiers. Vous pouvez utiliser au plus 3 variables x,y,z.

Objectif : si je donne des valeurs initiales de x,y et z par exemple 2, 3 et 4 votre voisin doit être incapable de trouver la valeur de x à la fin du programme.

Objectif avancé : je dois aussi être incapable de trouver la valeur de x.

Objectif très avancé : un ordinateur ne doit pas être capable de calculer x en moins de une minute.

L'âge du capitaine

Écrire un algorithme de 8 lignes au plus utilisant uniquement comme opération de base des sommes, des produits et des comparaisons d'entiers. Vous pouvez utiliser au plus 3 variables x,y,z.

Objectif : si je donne des valeurs initiales de x,y et z par exemple 2, 3 et 4 votre voisin doit être incapable de trouver la valeur de x à la fin du programme.

Objectif avancé : je dois aussi être incapable de trouver la valeur de x.

Objectif très avancé : un ordinateur ne doit pas être capable de calculer x en moins de une minute.