Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 3 : preuve d'algorithmes

Benjamin Wack

UFR IM²AG
UNIVERSITÉ Grenoble Alpes

2017- 2018

La dernière fois

- Écriture d'algorithmes récursifs
- Coût d'un algorithme récursif
- Complexité en moyenne
- Tri rapide

Aujourd'hui

- Spécification et correction d'un algorithme
- Terminaison d'un algorithme
- Drapeau Hollandais

Plan

Les besoins

- Quantité astronomique de code en circulation ou en développement
  (Google Chrome ou serveur World of Warcraft : 6 M lignes)
  (Windows 7 ou Microsoft Office 2013 : 40 M lignes)
  `http://www.informationisbeautiful.net`
- Omniprésence dans des systèmes critiques : finance, transports, économie, santé...
- Il faut pouvoir prouver qu'un programme s'exécute correctement dans toutes les situations

Mais correct selon quels critères ? Quelles situations à considérer ?

- Spécification
  - des données acceptables
  - du résultat attendu (généralement en fonction des données)
- exprimée dans un langage formel, généralement une propriété logique des données et du résultat.
- ne décrit **pas comment** fonctionne le programme.

Les propriétés recherchées

Terminaison

L'exécution de l'algorithme produit-elle un résultat en temps fini **quelles que soient** les données fournies ?

Correction partielle

Lorsque l'algorithme s'arrête, le résultat calculé est-il la solution cherchée quelles que soient les données fournies ?

Terminaison + correction partielle = correction totale

Quelles que soient les données fournies, l'algorithme s'arrête et donne une réponse correcte.

Pour certains problèmes il n'existe **que** des algorithmes **partiellement** corrects !

Une première écriture formelle

Soit un problème instancié par une donnée D et dont la réponse est fournie par un résultat R.

Une spécification peut être donnée sous la forme de :

- une propriété $P(D)$ de la donnée *(précondition)*;
- une propriété $Q(D, R)$ de la donnée et du résultat *(postcondition)*.

Un programme satisfait cette spécification si :

Pour toute donnée D qui vérifie la propriété P,
l'exécution du programme donne un résultat R qui vérifie $Q(D, R)$.

Le programme est alors dit correct *par rapport à* cette spécification.

Division euclidienne

Division par soustractions

DIV(a, b)
Données : Deux entiers a et b
Résultat : Le quotient q et le reste r de la division euclidienne de a par b

$r := a$
$q := 0$
while $r \geq b$
    $r := r - b$
    $q := q + 1$
return q, r

Données acceptables

- $b \neq 0$ sinon le problème n'a pas de sens (et la boucle non plus)
- $b > 0$ (?)
- $a > 0$ (?)

Division euclidienne

Division par soustractions

DIV(a, b)
Données : Deux entiers a et b
Résultat : Le quotient q et le reste r de la division euclidienne de a par b
$r := a$
$q := 0$
while $r \geq b$
  $r := r - b$
  $q := q + 1$
return q, r

Résultat attendu

- $a = q \times b + r$
- $0 \leq r < b$
- a et b inchangés

Le langage de programmation

Afin de pouvoir raisonner formellement, on fixe une syntaxe restreinte sur le langage de « programmation » utilisé :

- expressions
  $$E ::= \textit{variables} \mid \textit{constantes} \mid E + E \mid E * E \mid \ldots$$
- expressions booléennes
  $$B = B \texttt{ and } B \mid B \texttt{ or } B \mid \cdots \mid E = E \mid E < E \mid \ldots$$
- une instruction I peut être :
  - une affectation $\texttt{x} := E$
  - une séquence I_1 ; I_2
  - une conditionnelle `if` B `then` I_1 `else` I_2
  - une boucle `while` B `do` I

On peut alors raisonner **par cas** et **par induction** sur la forme du programme considéré.

- Pas de *for* (cf *while*)
- Pas de structures de données (mais possibilité d'étendre le langage)
- Pas d'appel de fonction (et donc pas de récursivité)

Le langage des propriétés

Les propriétés que nous exprimons à propos des données et des résultats sont généralement des **formules de logique du premier ordre** :

- connecteurs logiques $\neg$, $\wedge$, $\vee$, $\Rightarrow$
- quantificateurs $\forall$, $\exists$
- opérations et prédicats usuels sur les données ($+$, $*$, $<$, $=$...)

La plupart des variables sont partagées par le programme et les propriétés, certaines ne sont utilisées que dans les propriétés.

Attention

Une propriété des variables peut être vraie ou fausse à un point donné de l'exécution d'un programme, et selon les données initiales.

Ne pas confondre

- expression booléenne : évaluée dans une exécution du programme
- propriété : utilisée dans la démonstration

Ne pas confondre

- assertion : utilisée dans le test de programme, évaluée systématiquement et qui lève une exception si elle est fausse
- propriétés : (parfois appelées assertions !) sans se prononcer *a priori* sur leur valeur de vérité

Notion d'invariant

Idée de la démonstration d'un algorithme

De proche en proche, établir que la postcondition est vraie à chaque fois que la précondition est vraie.

- Affectation, séquence, condition :
  pas de vrai problème si la spécification est correctement écrite.
- Problème : la boucle
  (peut recevoir ses données d'une itération précédente)

Invariant

Un invariant est une propriété P des variables en début de boucle telle que

si P est vérifiée à une itération, alors elle l'est à l'itération suivante.

Méthodologie

1. **Choisir et exprimer** un invariant *judicieux*

   Pas de méthode systématique

2. **Démontrer** qu'il est vérifié avant d'entrer dans la boucle

   Utiliser les préconditions

3. **Démontrer** que s'il est vérifié au début d'une itération quelconque, il l'est aussi au début de l'itération suivante.

   Utiliser le corps de la boucle

   On note x' la valeur de x en fin de boucle

4. **Instancier** l'invariant en sortie de boucle et en déduire une postcondition.

   *Utiliser (la négation de) la condition du **while***

Annotation de la division euclidienne

Division par soustractions

DIV(a, b)
Données : Deux entiers a et b
Résultat : Le quotient q et le reste r de la division euclidienne de a par b
Précondition : $a \geq 0$ et $b > 0$

$r := a$
$q := 0$
while $r \geq b$ $\{invariant : \mathtt{a} = \mathtt{b} \times \mathtt{q} + \mathtt{r}\}$
  $r := r - b$
  $q := q + 1$
  $\{\mathtt{b} \times \mathtt{q}' + \mathtt{r}' = \mathtt{b} \times (\mathtt{q}+1) + (\mathtt{r}-\mathtt{b}) = \mathtt{b} \times \mathtt{q} + \mathtt{b} - \mathtt{b} + \mathtt{r} = \mathtt{b} \times \mathtt{q} + \mathtt{r} = \mathtt{a}\}$
return q, r

Postconditions : $\mathtt{a} = \mathtt{b} \times \mathtt{q} + \mathtt{r}$
$0 \leq \mathtt{r} < \mathtt{b}$
a et b inchangés

Exercice

Quel(s) programme(s) calcule(nt) la factorielle de n dans la variable F ?

A

```
i := 0
F := 1
while i <= n
    i := i + 1
    F := F * i
```

C

```
i := 1
F := 1
while i <= n
    F := F * i
    i := i + 1
```

B

```
i := 0
F := 1
while i < n
    i := i + 1
    F := F * i
```

D

```
i := 0
F := 1
while i < n
    F := F * i
    i := i + 1
```

Programme A

i := 0
F := 1 $\{F = 1 = 0! = i!\ \}$
while $i <= n$
$\{F = i!\}$
i := i + 1
F := F * i
$\{F' = F * i' = F * (i+1) = i! * (i+1) = (i+1)! \text{ donc } F' = i'!\}$

Invariant

$$F = i!$$

Mais en sortie de boucle $i = n + 1$ d'où $F = (n+1)!$

Programme B!

i := 0
F := 1 $\{F = 1 = 0! = i!\}$
while $i < n$
$\{F = i!\}$
i := i + 1
F := F * i $\{F' = F * i' = i! * (i+1) = (i+1)!$ donc $F' = i'!\}$

Invariant

$$F = i!$$

En sortie de boucle $\{i = n$ d'où $F = n!\}$

Programme C!

```
i := 1
F := 1    {F = 1 = 0! = (1 − 1)! = (i − 1)!}
while i <= n
    {F = (i − 1)!}
  F := F * i
  i := i + 1    {F' = F ∗ i = (i − 1)! ∗ i = i! donc
                 F' = (i + 1 − 1)! = (i' − 1)!}
```

Invariant

$F = i!$ ~~$F = i!$~~ alors en fin d'itération $F' = F * i = i! * i \neq i'!$: NON

$$F = (i-1)!$$

En sortie de boucle i = n+1 d'où F= n !

Programme D

```
i := 0
F := 1
while i < n
    F := F * i
    i := i + 1
```

Invariant

$F = (i-1)!$ $\cancel{F = (i-1)!}$ correctement propagé par la boucle
mais faux à l'entrée de la boucle : NON

En réalité on a toujours $F = 0$ après la 1è itération.

Correction partielle ou totale

Correction partielle

Pour toute donnée D qui vérifie la précondition P,
si le programme se termine,
alors son exécution donne un résultat R qui vérifie $Q(D, R)$.

Correction totale

Pour toute donnée D qui vérifie la précondition P,
l'exécution du programme se termine
et donne un résultat R qui vérifie $Q(D, R)$.

$$\text{Correction partielle} \wedge \text{terminaison} \Rightarrow \text{Correction totale}$$

D'où l'idée de prouver la terminaison **en même temps** que la correction partielle : on enrichit les annotations existantes.

Variant de boucle

Variant de boucle

Un **variant de boucle** est une expression :

- entière
- positive
- qui décroît **strictement** à chaque itération

Variants usuels

- i pour une boucle du type **for** $i = n$ **downto** 1
- $n - i$ pour une boucle du type **for** $i = 1$ **to** n
- $j - i$ pour deux variables i croissante et j décroissante
- ...
- mais pas de technique « systématique »

Variant de la division euclidienne

Division par soustractions

DIV(a, b)
Données : Deux entiers a et b
Résultat : Le quotient q et le reste r de la division euclidienne de a par b
Précondition : $a \geq 0$ et $b > 0$
$r := a$
$q := 0$
while $r \geq b$ $\{\mathtt{r} = n\}$
    $r := r - b$
    $q := q + 1$ $\{0 \leq \mathtt{r} < n\}$
return q, r

- r est clairement un variant de boucle
- On peut le formuler dans les annotations existantes grâce à une nouvelle variable logique.
  (n est quantifiée existentiellement de façon implicite)
- La preuve de $\mathtt{r} < n$ en fin de boucle repose sur la précondition $b > 0$ et sur la condition du `while`.

Les difficultés

Précondition et postcondition

- généralement faciles à écrire si le problème est correctement spécifié
- éventuellement nécessaire de renforcer la précondition si la preuve n'aboutit pas
- attention aux postconditions trop faibles

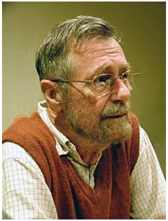
Invariants

- incluent souvent une généralisation de la postcondition
- demandent une compréhension fine de l'algorithme
- les trouver peut même précéder l'écriture de l'algorithme

Variants

- souvent immédiats, mais des cas particuliers très difficiles
- nécessitent parfois de s'appuyer sur les autres assertions

Le drapeau hollandais (1976)



E.W. Dijkstra (1930-2002)

- un des fondateurs de la science informatique
- algorithme de recherche de plus court chemin
- pile de récursivité pour ALGOL-60
- Turing Award 1972

Tableau de *n* éléments, chacun coloré en bleu, blanc ou rouge.

Objectif : réorganiser le tableau pour que :

- les éléments bleus soient sur la partie gauche
- les éléments blancs au centre
- les rouges en fin de tableau

1	2	3	4	5	6	7	8
A	B	C	D	E	F	G	H

→

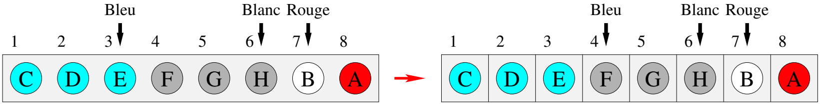
1	2	3	4	5	6	7	8
H	G	D	B	E	F	C	A

Contrainte : utiliser un minimum de mémoire supplémentaire (en place)

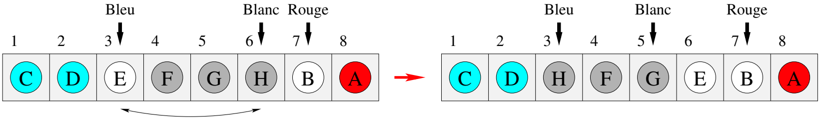
Les trois cas

Trois indices mémorisant où placer le prochain élément de chaque couleur

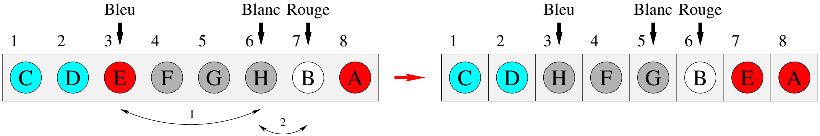
- Cas bleu



- Cas blanc



- Cas rouge



L'algorithme

DRAPEAU(T)

Données : Un tableau T de N éléments colorés (couleur C_1, C_2 et C_3)
Résultat : T contient les mêmes éléments rangés par couleur croissante

```
i₁ = 1
i₂ = N                          // iₖ est l'indice de la place du
i₃ = N                          // prochain élément de couleur Cₖ
while i₁ ≤ i₂
    switch Couleur (T[i₁]) do
        case C₁
            i₁ = i₁ + 1                    // l'élément est en place
        case C₂
            Échange (i₁, i₂)     // on le place en bonne position
            i₂ = i₂ − 1
        case C₃
            Échange (i₁, i₂)               // permutation circulaire
            Échange (i₂, i₃)                // pour libérer une case
            i₂ = i₂ − 1 ; i₃ = i₃ − 1
```

Complexité

- Nombre d'appels à la fonction `Couleur` = nombre d'itérations = N

$$\text{Coût}_{\texttt{Couleur}}(N) = N$$

  On évalue la couleur de chaque élément une et une seule fois.
- Nombre d'appels à la fonction `Échange` = somme du nombre d'échanges réalisés à chaque itération (0, 1 ou 2) :

$$0 \leq \text{Coût}_{\texttt{Échange}}(N) \leq 2N.$$

- Meilleur cas = tableau rempli d'éléments bleus
- Pire cas = tableau rempli d'éléments rouges

La complexité de l'algorithme en nombre d'échanges est donc au pire en $\mathcal{O}(N)$ et au mieux en $\mathcal{O}(1)$.

Et en moyenne ? Ça dépend de la distribution des données !

Éléments de démonstration

$i_1 = 1$; $i_2 = N$; $i_3 = N$
while $i_1 \leq i_2$
  switch `Couleur` $(T[i_1])$ **do**
    case C_1
      $i_1 = i_1 + 1$
    case C_2
      `Échange` (i_1, i_2)
      $i_2 = i_2 - 1$
    case C_3
      `Échange` (i_1, i_3)
      `Échange` (i_1, i_2)
      $i_2 = i_2 - 1$; $i_3 = i_3 - 1$

Terminaison : $i_2 - i_1$ est un variant acceptable

- À chaque itération i_1 augmente (cas 1) ou i_2 diminue (cas 2 et 3).
- La condition du **while** assure que $i_1 \leq i_2$.

$i_1 = 1$; $i_2 = N$; $i_3 = N$
while $i_1 \leq i_2$
  switch `Couleur` $(T[i_1])$ **do**
    case C_1
      $i_1 = i_1 + 1$
    case C_2
      `Échange` (i_1, i_2)
      $i_2 = i_2 - 1$
    case C_3
      `Échange` (i_1, i_3)
      `Échange` (i_1, i_2)
      $i_2 = i_2 - 1$; $i_3 = i_3 - 1$

Invariant de boucle

- T contient une permutation des éléments du tableau initial.
- Les éléments de 1 à $i_1 - 1$ sont de couleur C_1.
- Les éléments de $i_2 + 1$ à i_3 sont de couleur C_2.
- Les éléments de $i_3 + 1$ à N sont de couleur C_3.

Démonstration

Préservation des éléments du tableau

Garantie par l'utilisation exclusive de la procédure **Échange**.

(mais cette propriété doit faire partie de la spécification de **Échange** !)

Rangement des couleurs

- À l'entrée dans la boucle cette propriété ne concerne aucun élément.
- Au cours d'une itération :

  cas 1 : un élément C_1 est placé, les autres sont inchangés
  cas 2 : un élément C_2 est placé, les autres sont inchangés
  cas 3 : un élément C_3 est placé, les éléments C_2 sont décalés, les autres sont inchangés
- En sortie de boucle $i_2 = i_1 - 1$ donc :
  - Les éléments de 1 à $i_1 - 1$ sont de couleur C_1.
  - Les éléments de i_1 à i_3 sont de couleur C_2.
  - Les éléments de $i_3 + 1$ à N sont de couleur C_3.

Variantes

Même problème avec :

- 2 couleurs seulement *(c'est **Partition** pour le tri rapide)*
- plus de 3 couleurs *(cf TD2)*

Tri rapide avec plusieurs pivots :

- Remplacer **Partition** par le « drapeau » approprié
- Autant d'appels récursifs que de sous-tableaux formés
- Mais au final pas de gain (voire une perte) d'efficacité

En résumé

Aujourd'hui

- Un algorithme est démontré correct **par rapport à une spécification**
- Un invariant est une **propriété** préservée par une boucle, utile pour démontrer la correction de l'algorithme
- Un variant est une quantité qui **décroît** à chaque itération d'une boucle et assure sa terminaison
- Drapeau hollandais

La prochaine fois

- Logique de Hoare
- Annotation de programmes

Test ou preuve ?

"Testing shows the presence, not the absence of bugs"
E. W. Dijkstra, 1930-2002

Le test :

- valide une **implantation** plutôt qu'un algorithme
- permet des vérifications rapides
- peut être utilisé en cours de développement
- fait apparaître les limites du modèle

La preuve :

- fournit une **garantie** incontestable sur le fond de l'algorithme
- mais n'élimine pas (complètement) les erreurs de programmation
- nécessite des outils formels pour une utilisation à grande échelle