

Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 3 : preuve d'algorithmes

Benjamin Wack



2021 – 2022

La dernière fois

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

Aujourd'hui

- ▶ Invariant, précondition, postcondition
- ▶ Spécification et correction d'un algorithme
- ▶ Terminaison d'un algorithme
- ▶ Drapeau Hollandais

Plan

Preuve de correction d'un algorithme

- Spécification formelle

- Annotations de programmes

Terminaison d'un algorithme

Drapeau Hollandais

- Le problème et l'algorithme

- Analyse de l'algorithme

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 précondition P ,
le résultat R produit par le programme vérifie la postcondition Q .

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

Division euclidienne

Division par soustractions

$\text{DIV}(a, b)$

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

Notion d'invariant

Idee de la démonstration d'un algorithme

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

(Interlude : tour de cartes)

- ▶ 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* : $a = b \times q + r$ }

$r := r - b$

$q := q + 1$

 { $b \times q' + r' = b \times (q + 1) + (r - b) = b \times q + b - b + r = b \times q + r$ }
 { et par hypothèse de l'invariant $b \times q + r = a$ }
 }

return q, r

Postconditions : $a = b \times q + r$

$0 \leq r < b$ \leftarrow (reste à démontrer)

a et b inchangés \leftarrow (reste à démontrer)

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
```

B

```
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
```

D

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

Programme A

$i := 0$

$F := 1 \quad \{F = 1 = 0! = i!\}$

while $i \leq n$

$\{F = i!\}$

$i := i + 1$

$F := F * i$

$\{F' = F * i' = F * (i + 1) = i! * (i + 1) = (i + 1)! \text{ donc } F' = i'!\}$

Invariant

$$F = i!$$

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

Programme $B!$

$i := 0$

$F := 1 \quad \{F = 1 = 0! = i!\}$

while $i < n$

$\{F = i!\}$

$i := i + 1$

$F := F * i \quad \{F' = F * i' = i! * (i + 1) = (i + 1)! \text{ donc } F' = i'!\}$

Invariant

$$F = i!$$

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

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

$F = i! \text{ et } i = i$ 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

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

En réalité on a toujours $F = 0$ après la 1^è 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 la postcondition Q .

Correction **totale**

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

SI correction partielle ET terminaison ALORS correction totale

Variante de boucle

Variante 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 »

Variante 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 $r \geq b$

┌ $r := r - b$
└ $q := q + 1$

return q, r

r est clairement un variant de boucle

- ▶ il est entier
- ▶ il décroît strictement à chaque itération (car $b > 0$)
- ▶ il est toujours positif (car $b > 0$ et 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 (*exemple du tri*)

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

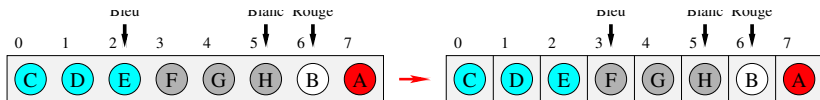


Contrainte : utiliser un minimum de mémoire supplémentaire (en place)

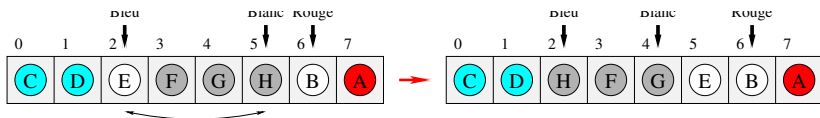
Les trois cas

Trois indices mémorisant où placer le prochain élément de chaque couleur

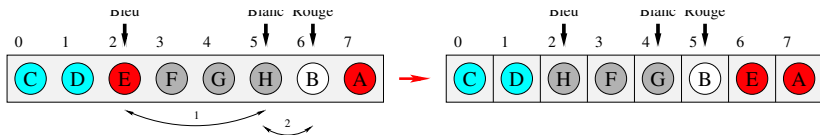
► Cas où E est bleu



► Cas où E est blanc



► Cas où E est rouge



L'algorithme

DRAPEAU(T)

Données : Un tableau T de N éléments colorés (*bleu*, *blanc* ou *rouge*)

Résultat : T contient les mêmes éléments rangés par couleur croissante

```

ibleu = 0
iblanc =  $N - 1$            // ik est l'indice de la place du
irouge =  $N - 1$          // prochain élément de couleur k
while ibleu ≤ iblanc
    switch Couleur ( $T[i_{bleu}]$ ) do
        case bleu do
            | ibleu = ibleu + 1           // l'élément est en place
        case blanc do
            | Échange (ibleu, iblanc) // on le place en bonne position
            | iblanc = iblanc - 1
        case rouge do
            | Échange (ibleu, iblanc)           // permutation circulaire
            | Échange (iblanc, irouge)           // pour libérer une case
            | iblanc = iblanc - 1; irouge = irouge - 1

```

Complexité

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

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

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) :

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

- ▶ Cas favorable = tableau rempli d'éléments bleus
- ▶ Cas défavorable = 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_bleu = 0 ; i_blanc = N - 1 ; i_rouge = N - 1
while i_bleu ≤ i_blanc
  switch Couleur (T[i_bleu]) do
    case bleu do
      | i_bleu = i_bleu + 1
    case blanc do
      | Échange (i_bleu, i_blanc)
      | i_blanc = i_blanc - 1
    case rouge do
      | Échange (i_bleu, i_rouge)
      | Échange (i_bleu, i_blanc)
      | i_blanc = i_blanc - 1 ; i_rouge = i_rouge - 1
  
```

Terminaison : $i_{blanc} - i_{bleu}$ est un variant acceptable

- ▶ À chaque itération i_{bleu} augmente (cas 1) ou i_{blanc} diminue (cas 2 et 3).
- ▶ La condition du **while** assure que $i_{bleu} \leq i_{blanc}$.

```

i_bleu = 0 ; i_blanc = N - 1 ; i_rouge = N - 1
while i_bleu ≤ i_blanc
  switch Couleur (T[i_bleu]) do
    case bleu do
      | i_bleu = i_bleu + 1
    case blanc do
      | Échange (i_bleu, i_blanc)
      | i_blanc = i_blanc - 1
    case rouge do
      | Échange (i_bleu, i_rouge)
      | Échange (i_bleu, i_blanc)
      | i_blanc = i_blanc - 1 ; i_rouge = i_rouge - 1

```

Invariant de boucle

- ▶ T contient une permutation des éléments du tableau initial.
- ▶ Les éléments de 0 à $i_{bleu} - 1$ sont de couleur *bleu*.
- ▶ Les éléments de $i_{blanc} + 1$ à i_{rouge} sont de couleur *blanc*.
- ▶ Les éléments de $i_{rouge} + 1$ à $N - 1$ sont de couleur *rouge*.

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. (donc elle est vraie!)
- ▶ Au cours d'une itération :
 - cas 1 : un élément *bleu* est placé, les autres sont inchangés
 - cas 2 : un élément *blanc* est placé, les autres sont inchangés
 - cas 3 : un élément *rouge* est placé, les éléments *blanc* sont décalés, les autres sont inchangés
- ▶ En sortie de boucle $i_{blanc} = i_{bleu} - 1$ donc :
 - ▶ Les éléments de 0 à $i_{bleu} - 1$ sont de couleur *bleu*.
 - ▶ Les éléments de i_{bleu} à i_{rouge} sont de couleur *blanc*.
 - ▶ Les éléments de $i_{rouge} + 1$ à $N - 1$ sont de couleur *rouge*.

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 **décroit** 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

Le test :

- ▶ valide une **implantation** plutôt qu'un algorithme
- ▶ permet rapidement d'éliminer des « bugs »
- ▶ 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