

Algorithmique et Analyse d'Algorithmes

L3 Info

Cours 6 : preuves avancées d'algorithmes

Benjamin Wack



2025 – 2026

La dernière fois

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

Aujourd'hui

- ▶ Preuve d'algorithme récursif
- ▶ Logique de Hoare
- ▶ Dichotomie

Partons d'un exemple (TD1)

Pour un arbre binaire A , écrivez une fonction qui calcule la hauteur de la feuille la plus haute (la feuille la plus proche de la racine)

FeuilleHaute(A)

```
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille  
  └ Renvoyer 0  
sinon  
  └ Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Quels problèmes peut-on rencontrer dans l'exécution d'un tel algorithme ?

- ▶ appels récursifs infinis → **terminaison**
- ▶ appels récursifs mal définis → **préconditions**
- ▶ résultats des appels récursifs incorrects → **postconditions**
- ▶ résultat de l'appel principal incorrect → **preuve de la fonction**

Terminaison des fonctions récursives

FeuilleHaute(A)

```
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille
└ Renvoyer 0
sinon
└ Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Méthode

Comme pour la terminaison des boucles, exhiber un **variant** (entier positif décroissant strictement)

À adapter dans le cas des fonctions récursives :

- ▶ le variant est calculé à partir des **arguments** de la fonction
- ▶ il doit décroître dans **chaque appel** récursif

Dans cet exemple : le **nombre de nœuds** de l'arbre décroît strictement dans chaque appel récursif.

Préconditions

```
FeuilleHaute(A)
```

```
si estVide(A)
```

```
  | Renvoyer -1
```

```
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille
```

```
  | Renvoyer 0
```

```
sinon
```

```
  | Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Les appels récursifs de cette fonction sont incorrects car on a une **précondition implicite** : A doit être non vide.

Comment rectifier ce problème ?

- ▶ Ajouter des cas dans l'algorithme pour alléger la précondition
OU
- ▶ Éviter les appels sur des données qui ne respectent pas la précondition

Postconditions

FeuilleHaute(A)

```
si estVide(A)
  └ Renvoyer -1
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille
  └ Renvoyer 0
sinon
  └ Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Postconditions de cette fonction :

- ▶ L'arbre A possède une feuille à la **profondeur renvoyée**
- ▶ Toutes les feuilles de A ont une profondeur \geq à la **valeur renvoyée**
- ▶ Et le cas où on renvoie -1 ? **on a rendu la spécification difficile à exprimer**

Algorithme v3

FeuilleHaute(A)

```
si estVide(A)
  └ Renvoyer -1
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille
  └ Renvoyer 0
sinon si estVide(FG(A))
  └ Renvoyer 1 + FeuilleHaute(FD(A))
sinon si estVide(FD(A))
  └ Renvoyer 1 + FeuilleHaute(FG(A))
sinon
  └ Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Préconditions

L'arbre A est non vide

Vérifier aussi qu'on n'a pas cassé la terminaison !

Postconditions

- ▶ L'arbre A possède une feuille à la profondeur renvoyée
- ▶ Toutes les feuilles de A ont une profondeur $>$ à la valeur renvoyée

Preuve de la fonction

FeuilleHaute(A)

```
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille
└ Renvoyer 0
sinon si estVide(FG(A))
└ Renvoyer 1 + FeuilleHaute(FD(A))
sinon si estVide(FD(A))
└ Renvoyer 1 + FeuilleHaute(FG(A))
sinon
└ Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Il faut démontrer ces postconditions :

Dans les cas de base, directement

- ▶ A possède une feuille à la profondeur renvoyée :
c'est A
- ▶ Toutes les feuilles de A ont une profondeur \geq à la valeur renvoyée :
il n'y en a pas d'autre

Preuve de la fonction

FeuilleHaute(A)

```
si estVide(FG(A)) et estVide(FD(A))           // A est une feuille
└ Renvoyer 0
sinon si estVide(FG(A))
└ Renvoyer 1 + FeuilleHaute(FD(A))
sinon si estVide(FD(A))
└ Renvoyer 1 + FeuilleHaute(FG(A))
sinon
└ Renvoyer 1 + minimum(FeuilleHaute(FG(A)), FeuilleHaute(FD(A)))
```

Il faut démontrer ces postconditions :

Dans les cas récursifs **en supposant les appels récursifs corrects**

- ▶ A possède une feuille à la profondeur renvoyée :
en utilisant l'hypothèse sur l'appel récursif sélectionné
- ▶ Toutes les feuilles de A ont une profondeur \geq à la valeur renvoyée :
en utilisant l'hypothèse sur les **deux** appels récursifs

Méthode de preuve des algorithmes récursifs

Pour démontrer qu'une fonction récursive est correcte :

- ▶ Exhiber un **variant** calculé à partir de ses arguments (qui décroît dans **chaque appel récursif**)
- ▶ Spécifier les préconditions à propos des arguments et les postconditions à propos de la valeur renvoyée
- ▶ Vérifier que dans **chaque appel récursif** les **préconditions sont respectées**
- ▶ Démontrer que les **cas de base respectent les postconditions**
- ▶ Démontrer que l'appel principal **respecte les postconditions** en **supposant que les appels récursifs** sont corrects

Problème et intuition



Où est la frontière ?

Littéralement, *dichotomie* = « couper en deux »



Bleu/rouge = par exemple inférieur ou supérieur à un élément recherché,
mais pas seulement

Algorithme récursif

DICHOTOMIE(T, v, g, d)

Données : Un tableau T en ordre croissant
 Une valeur v
 Deux indices $0 \leq g, d < N$

Résultat : Un indice r tel que :

- si $r = -1$ alors $T[g..d] \neq v$
- sinon $T[r] = v$

```

si  $g > d$ 
  └ renvoyer -1
 $m := (g + d) / 2$ 
si  $T[m] = v$ 
  └ renvoyer  $m$ 
si  $T[m] < v$ 
  └ renvoyer DICHOTOMIE( $T, v, m + 1, d$ )
sinon //  $T[m] > v$ 
  └ renvoyer DICHOTOMIE( $T, v, g, m - 1$ )
  
```



J. Mauchly
 (version itérative)
 (1907-1980)

Démonstrons :

- ▶ Terminaison
- ▶ Correction

Terminaison

On a deux appels récursifs possibles :

- ▶ DICHOTOMIE($T, v, m + 1, d$)
- ▶ DICHOTOMIE($T, v, g, m - 1$)

avec à chaque fois $m = (g + d) / 2$ et $g \leq d$

On propose comme variant $d - g$:

- ▶ $d - (m + 1) = d - \frac{g+d}{2} - 1 = \frac{d-g}{2} - 1 < d - g$
- ▶ de même $(m - 1) - g = \frac{d-g}{2} - 1 < d - g$

On peut même en déduire la complexité de l'algorithme :

- ▶ À chaque appel récursif $d' - g' < \frac{d-g}{2}$
- ▶ Donc au bout de i appels récursifs imbriqués $d - g \leq \frac{N}{2^i}$
- ▶ Et lorsque $d - g = 0$ il reste au pire 1 appel récursif

Comme le corps de la fonction s'exécute en temps constant, le coût de l'algorithme est de l'ordre de i tel que $\frac{N}{2^i} < 1$, soit $\mathcal{O}(\log_2(N))$.

Correction

Cas de base

- ▶ Si on renvoie -1 alors $g > d$, clairement aucune valeur dans $T[g..d]$ ne peut être égale à v .
- ▶ Si on renvoie $m \neq -1$ alors c'est que $T[m] = v$ (condition du **si**)

Cas récurifs

- ▶ On suppose que $\text{DICHOTOMIE}(T, v, m + 1, d)$ respecte notre spécification.
- ▶ Il peut donc se présenter deux cas :
 - ▶ soit cet appel récursif renvoie un $r \neq -1$, alors **par hypothèse** $T[r] = v$ et on renvoie ce même indice
 - ▶ soit il renvoie -1 , alors **par hypothèse** $T[m + 1..d] \neq v$
 - ▶ or nous savons que $T[m] < v$ (condition du **si**) et que T est croissant donc $T[g..m] \leq T[m] < v$ (précondition)
 - ▶ d'où ce que nous devons prouver : $T[g..d] \neq v$
- ▶ Raisonnement similaire pour l'autre appel récursif

Motivations

La méthode des invariants fonctionne mais ne passe pas à l'échelle :

- ▶ l'invariant doit être « deviné »
- ▶ la démonstration elle-même n'est pas automatisable
- ▶ et elle n'est même pas *vérifiable*

Solution proposée :

- ▶ un système de déduction **formel**
- ▶ annotation du programme **en partant de la fin**

- ▶ Progrès vers l'automatisation
- ▶ Pas totale cependant : la preuve d'algorithme est un problème **indécidable**
- ▶ Possibilité d'utiliser des *assistants de démonstration* (Why3 par exemple)

Le langage de programmation

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

- ▶ une instruction I peut être :
 - ▶ une affectation $x := E$
 - ▶ une séquence $I_1 ; I_2$
 - ▶ une conditionnelle $\text{if } B \text{ then } I_1 \text{ else } I_2$
 - ▶ une boucle $\text{while } B \text{ do } I$

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

- ▶ Pas de *for* (utiliser *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 *non*, *et*, *ou*, \Rightarrow
- ▶ quantificateurs \forall , \exists
- ▶ opérations et prédicats usuels sur les données ($+$, $*$, $<$, $=\dots$)

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

Triplet de Hoare

Tony Hoare (1934-)

- ▶ Tri rapide
- ▶ Compilateur pour ALGOL-60
- ▶ Spécification formelle de langages (y compris multi-processus)
- ▶ Turing Award 1980

On travaille sur le langage minimal défini la semaine dernière (affectation, séquence, conditionnelle, boucle `while` + quelques opérations).

Triplet de Hoare

Un triplet de Hoare est de la forme $\{P\} I \{Q\}$ où :

- ▶ P et Q sont des formules logiques comportant éventuellement des variables de I
- ▶ I est une instruction de notre langage de programmation

Nota Bene : dans sa présentation originale, Hoare écrivait plutôt $P \{I\} Q$, ce qu'on retrouve dans certains documents.

L'intention derrière le système

Le triplet $\{P\} I \{Q\}$ exprime la propriété suivante :

Si P (précondition) est vérifiée avant l'exécution de I ,
alors Q (postcondition) est vérifiée après son exécution.

Quelques triplets

$\{true\} x := 3 \{x = 3\}$

$\{x > 0\} x := x + 1 \{x > 0\}$

$\{x > 0\} x := x - 1 \{x < 2\}$

$\{x = n\} x := x + 1 \{x = n + 1\}$

$\{true\} \text{if } x < 0 \text{ then } x := -x \text{ else skip } \{x > 0\}$

Un tel triplet peut être **valide** ou **non**.

Validité et démonstration

Le triplet $\{P\} I \{Q\}$ est dit **valide** si :

quel que soit l'état mémoire initial qui vérifie P ,
l'exécution de I produit un état mémoire qui vérifie Q .

Problème : il y a une infinité d'états mémoire à considérer.

On construit donc les triplets de Hoare à l'aide d'un **système de déduction**, qui ne se base **que** sur une décomposition de I .

Théorème de correction de la logique de Hoare (admis)

Si

on peut déduire un triplet $\{P\} I \{Q\}$ dans le système de Hoare

alors

pour tout état mémoire qui vérifie P , l'exécution de I produit un état mémoire qui vérifie Q .

Système formel de déduction

Une **règle** est constituée :

- ▶ d'**hypothèses** ou **prémisses** H_1, \dots, H_n
- ▶ d'une **conclusion**
- ▶ éventuellement on étiquette une règle par son nom

$$\frac{H_1 \dots H_n}{C} R$$

Exemple : Introduction de la conjonction $\frac{A \quad B}{A \text{ et } B}$

Système de déduction = ensemble fini, fixé de règles

Preuve dans un système formel

Un arbre de preuve est formé de plusieurs règles, les conclusions des unes formant les prémisses des autres :

$$\frac{\frac{H_1}{\quad} \quad \dots \quad \frac{H_n}{\quad}}{\quad \dots \quad} \quad \text{prouve } A \text{ sous les hypothèses } H_1, \dots, H_n.$$

- ▶ Un **axiome** est une règle sans prémisses.
- ▶ Un **arbre complet** est un arbre de preuve dont toutes les feuilles sont des axiomes.

Démontrer A dans le système choisi, c'est exhiber un arbre complet dont la conclusion est A .

Une règle simple : la séquence

$$\frac{\{P\} I \{Q\} \quad \{Q\} J \{R\}}{\{P\} I ; J \{R\}}$$

$$\vdots$$

$$\vdots$$

$$\frac{\{x > 0\} x := x + 1 \{x > 1\} \quad \{x > 1\} y := 2 * x \{y > 2\}}{\{x > 0\} x := x + 1 ; y := 2 * x \{y > 2\}}$$

Attention

L'arbre de preuve ainsi construit n'est pas encore complet.

Autre règle simple : la conditionnelle

$$\frac{\{P \text{ et } C\} I \{Q\} \quad \{P \text{ et non } C\} J \{Q\}}{\{P\} \text{ if } C \text{ then } I \text{ else } J \{Q\}}$$

$$\frac{\{y \geq 0\} x := y \{x \geq 0\} \quad \{y < 0\} x := 0 \{x \geq 0\}}{\{true\} \text{ if } y \geq 0 \text{ then } x := y \text{ else } x := 0 \{x \geq 0\}}$$

Remarque : on prouve la **même** chose sous les **mêmes** hypothèses (excepté la condition booléenne) : **pas réaliste en général**.

L'affectation : ça se complique

Quelle règle pour $x := b + a$?

- ▶ postcondition $\{x = b + a\}$:
oui mais ne tient pas compte d'une éventuelle précondition
- ▶ si la précondition est $\{z > 0\}$: **conserver** cette information
- ▶ si la précondition est $\{x = 3\}$: **oublier** cette information
- ▶ si la précondition est $\{b = 0\}$: on voudrait en **déduire** $x = a$

Plus compliqué : $x := x + 2$

- ▶ postcondition $\{x = x + 2\}$: **n'a pas de sens**
- ▶ si la précondition est $\{x = 4\}$: on voudrait **déduire** $\{x = 6\}$
- ▶ si la précondition est $\{x > 0\}$: on voudrait **déduire** $\{x > 2\}$

Vers une règle pour l'affectation

Ce qu'on voudrait

- ▶ Établir un lien **logique** entre la précondition et la postcondition
- ▶ **Exploiter** le changement de valeur de la variable x affectée

Idée :

- ▶ Utiliser une même propriété P en précondition et en postcondition
- ▶ **Substituer** x par sa nouvelle valeur

~~$\{P\} x := E \{P[x \leftarrow E]\}$~~ : **Cette règle est incorrecte**

Essai sur un exemple

- ▶ Soit l'instruction $x := x - 1$
- ▶ Soit la précondition $P : x \geq 0$

Alors cette règle donne $\{x \geq 0\} x := x - 1 \{x \geq 1\}$

Règle de l'affectation (**axiome**)

$$\overline{\{Q[x \leftarrow E]\} x := E \{Q\}}$$

Raisonnement **arrière** : pour que Q soit vraie pour x après l'affectation, il faut qu'elle soit déjà vraie pour E avant l'affectation.

$$\overline{\{b + a = a\} x := b + a \{x = a\}}$$

$$\overline{\{x + 2 > 2\} x := x + 2 \{x > 2\}}$$

Deux remarques

- ▶ Il faut pouvoir manipuler les (in)égalités.
- ▶ Ne marche que si une variable est toujours manipulée explicitement par son nom : pas d'aliasing, pas de pointeurs, pas d'effets de bord.

Exemple : échange de deux variables

$$\{x = a \text{ et } y = b\} \quad t := x ; x := y ; y := t \quad \{x = b \text{ et } y = a\}$$

$$\frac{\frac{\{y = b \text{ et } x = a\} t := x \quad \{y = b \text{ et } t = a\} \quad \{y = b \text{ et } t = a\} x := y \quad \{x = b \text{ et } t = a\}}{\{x = a \text{ et } y = b\} t := x ; x := y \quad \{x = b \text{ et } t = a\}} \quad \{x = b \text{ et } t = a\} y := t \quad \{x = b \text{ et } y = a\}}{\{x = a \text{ et } y = b\} t := x ; x := y ; y := t \quad \{x = b \text{ et } y = a\}}$$

Règle du while

$$\frac{\{P \text{ et } C\} I \{P\}}{\{P\} \text{ while } C \text{ do } I \{P \text{ et } \textit{non } C\}}$$

(P est l'invariant de la boucle)

$$\frac{\frac{\overline{\{x < b\} x := x + 1 \{x \leq b\}}}{\{x \leq b \text{ et } x < b\} x := x + 1 \{x \leq b\}}}{\{x \leq b\} \text{ while } x < b \text{ do } x := x + 1 \{x \leq b \text{ et } x \geq b\}}$$

$$\frac{}{\{x \leq b\} \text{ while } x < b \text{ do } x := x + 1 \{x = b\}}$$

Règles logiques

Soit à démontrer :

$$\frac{\frac{\text{(axiome)}}{\{y \geq 0\} x := y \{x \geq 0\}} \quad \frac{\text{???}}{\{y < 0\} x := 1 \{x \geq 0\}}}{\{true\} \text{ if } y \geq 0 \text{ then } x := y \text{ else } x := 1 \{x \geq 0\}}$$

Renforcement de la précondition

$$\frac{P \Rightarrow P' \quad \{P'\} I \{Q\}}{\{P\} I \{Q\}}$$

$$\frac{y < 0 \Rightarrow 1 \geq 0 \quad \{1 \geq 0\} x := 1 \{x \geq 0\}}{\{y < 0\} x := 1 \{x \geq 0\}}$$

Et réciproquement...

Affaiblissement de la postcondition

$$\frac{\{P\} I \{Q\} \quad Q \Rightarrow Q'}{\{P\} I \{Q'\}}$$

Si $A \Rightarrow B$ la condition A est dite **plus forte** que B .
la condition B est **plus faible** que A .

Ces implications sont à **prouver** :

- ▶ à la main
- ▶ par d'autres règles d'inférence (déduction naturelle...)
- ▶ à l'aide d'un assistant de démonstration (logiciel)

En pratique

Le système de Hoare est complexe à utiliser :

- ▶ règles lourdes à appliquer
- ▶ taille des arbres de preuve
- ▶ mais il se prête bien à l'*automatisation* de la preuve

On en retiendra cependant quelques idées pour les « petites » preuves :

- ▶ Décomposer le programme par induction structurelle
- ▶ Annoter chaque étape du programme avec une propriété
- ▶ Remonter de la postcondition vers la précondition
- ▶ Exploiter *non C* en sortie du while

En résumé

Aujourd'hui

- ▶ Méthode de preuve des **algorithmes récursifs** :
 - ▶ **Variant** calculé à partir des arguments
 - ▶ Démontrer les postconditions **en supposant les appels récursifs corrects**
- ▶ Notion de **système formel** de preuve : **logique de Hoare** pour la spécification et la preuve de programme
- ▶ Méthode de preuve par **raisonnement arrière** : on part des postconditions pour construire les étapes précédentes

La prochaine fois

- ▶ Structure de données dynamique
- ▶ Arbre Binaire de Recherche