# Preuve de programmes

Judicaël Courant

Lycée La Martinière-Monplaisir, Lyon

4 mai 2017

# Une phrase sibylline

*Les invariants de boucles sont introduits pour s'assurer de la correction des segments itératifs.*

# Une phrase sibylline

*Les invariants de boucles sont introduits pour s'assurer de la correction des segments itératifs.*

Programme officiel du tronc commun

# Une phrase sibylline

*Les invariants de boucles sont introduits pour s'assurer de la correction des segments itératifs.*

Programme officiel du tronc commun

## But de cet exposé

- Donner un peu de contexte
- Montrer la mise en œuvre sur quelques exemples

# Plan

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

FirstEnergy, 2013 coupure électrique pour $5 \times 10^7$ personnes, pendant 7h à 2 jours, quelques morts

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

FirstEnergy, 2013 coupure électrique pour $5 \times 10^7$ personnes, pendant 7h à 2 jours, quelques morts

Therac 25, 1985-1987 5 morts avérées

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

FirstEnergy, 2013 coupure électrique pour $5 \times 10^7$ personnes, pendant 7h à 2 jours, quelques morts

Therac 25, 1985-1987 5 morts avérées

Ambulances de Londres, 2017 panne 5h (20-30 morts ?)

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

FirstEnergy, 2013 coupure électrique pour $5 \times 10^7$ personnes, pendant 7h à 2 jours, quelques morts

Therac 25, 1985-1987 5 morts avérées

Ambulances de Londres, 2017 panne 5h (20-30 morts ?)

Toyota, 2010 $10^2$ morts ?

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

FirstEnergy, 2013 coupure électrique pour $5 \times 10^7$ personnes, pendant 7h à 2 jours, quelques morts

Therac 25, 1985-1987 5 morts avérées

Ambulances de Londres, 2017 panne 5h (20-30 morts ?)

Toyota, 2010 $10^2$ morts ?

Satellite soviétique, 1983 détecte une attaque US

# Quelques bugs célèbres

Bank of Queensland, 2010 terminaux en panne une semaine

Mars Climate Orbiter, 1999 $3 \times 10^8$ \$

Knight Capital, 2012 $10^7$ \$ par minute pendant $45$ minutes

Vol 501 Ariane 5, 1996 $5 \times 10^8$ €

Bug de l'an 2000 $3 \times 10^{11}$ \$ dans le monde

FirstEnergy, 2013 coupure électrique pour $5 \times 10^7$ personnes, pendant 7h à 2 jours, quelques morts

Therac 25, 1985-1987 5 morts avérées

Ambulances de Londres, 2017 panne 5h (20-30 morts ?)

Toyota, 2010 $10^2$ morts ?

Satellite soviétique, 1983 détecte une attaque US

NORAD, 1980 détecte une attaque soviétique

# Bugs : tous les autres

- Ce n'est qu'une petite partie...

# Bugs : tous les autres

- Ce n'est qu'une petite partie...
- ... de la partie émergée de l'iceberg des problèmes de sûreté

# Bugs : tous les autres

- Ce n'est qu'une petite partie...
- ... de la partie émergée de l'iceberg des problèmes de sûreté
- Il y a aussi les problèmes de sécurité.

# Bugs : tous les autres

- Ce n'est qu'une petite partie...
- ... de la partie émergée de l'iceberg des problèmes de sûreté
- Il y a aussi les problèmes de sécurité.
- 500 postes d'agrégés d'informatique : $10^9$ € pour 40 ans de carrière + la retraite.

# Comment lutter contre les bugs

Pratiques d'ingénierie classique :

- Faire appel à des ingénieurs compétents :
  - dans le domaine concerné
  - et en informatique
- Rigueur dans la conception
- Tests

# Différence entre informatique et ingénierie classique

## Écroulement d'un édifice

Il est souvent possible

- d'anticiper (dégradation progressive)
- de rectifier avant la catastrophe (diagnostic possible)
- d'éviter une catastrophe d'ampleur (simultanéité improbable)

## Écroulement de systèmes informatique

Généralement :

- pas de signe avant-coureur
- écroulement simultané à grande échelle possible

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Besoin de plus que des tests

Les tests sont

Nécessaires : ils permettent de trouver des bugs

# Besoin de plus que des tests

Les tests sont

Nécessaires : ils permettent de trouver des bugs

Insuffisants : ils ne peuvent montrer l'absence de bugs

# Que sont les méthodes formelles ?

- Outils mathématiques/logiques pour démontrer la correction d'un logiciel
- Principalement appliquées aux logiciels critiques
- Automatisées à des degrés divers

# Une classe de méthodes formelles

## Sémantique axiomatique

Combinaison :

- d'un langage pour décrire le comportement des programmes
- de règles de raisonnement

## Logique de Hoare

Une sémantique axiomatique pour les programmes impératifs.

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Un mini-langage impératif

- instructions :

$$
\begin{array}{rll}
i ::= & \text{while } e \text{ do } i & \text{boucle} \\
| & \text{if } e \text{ then } i_1 \text{ else } i_2 & \text{conditionnelle} \\
| & v := e & \text{affectation} \\
| & i_1 ; \ i_2 & \text{séquence}
\end{array}
$$

- expressions :

$$
\begin{array}{rll}
e ::= & v & \text{variable} \\
| & e_1 \ op \ e_2 & \text{application d'une opération} \\
| & 0 \ | 1 \ | 2 \ | \ldots & \text{constantes entières}
\end{array}
$$

où $op \in \{ +, -, \times, /, \%, <, >, \leq, \geq, = \}$

# Triplets de Hoare

## Définition (Triplet de Hoare)

Triplet $\{P\}\, i\, \{Q\}$, où :

- $P$ formule mathématique (précondition)
- $i$ instruction
- $Q$ formule mathématique (postcondition)

## Définition (Correction partielle)

$\{P\}\, i\, \{Q\}$ valide si pour toute valeur initiale des variables vérifiant $P$ et telle que l'exécution de $i$ termine, la valeur des variables à la fin de l'exécution vérifie $Q$.

# Règles de la logique de Hoare

- On va se donner des règles de raisonnement (règles d'inférence) permettant de déduire des triplets de Hoare.
- On les introduit par des exemples.

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Échange de variables

Le programme suivant échange le contenu des variables $a$ et $b$ :

$$t := a;\ \ a := b;\ \ b := t$$

## Modélisation par un triplet de Hoare

Dire que le programme est correct, c'est dire que le triplet de Hoare suivant est valide

$$\left\{ \begin{array}{l} a = \alpha \\ b = \beta \end{array} \right\} t := a;\ \ a := b;\ \ b := t \left\{ \begin{array}{l} b = \alpha \\ a = \beta \end{array} \right\}$$

# Montrons la correction

De droite à gauche, on va montrer :

$$\left\{\begin{array}{l} t{=}\alpha \\ a{=}\beta \end{array}\right\} b{:=}t \left\{\begin{array}{l} b{=}\alpha \\ a{=}\beta \end{array}\right\} \qquad \left\{\begin{array}{l} t{=}\alpha \\ b{=}\beta \end{array}\right\} a{:=}b \left\{\begin{array}{l} t{=}\alpha \\ a{=}\beta \end{array}\right\}$$

$$\left\{\begin{array}{l} a{=}\alpha \\ b{=}\beta \end{array}\right\} t{:=}a \left\{\begin{array}{l} t{=}\alpha \\ b{=}\beta \end{array}\right\}$$

D'où $\left\{\begin{array}{l} a = \alpha \\ b = \beta \end{array}\right\} t := a;\ a := b;\ b := t \left\{\begin{array}{l} b = \alpha \\ a = \beta \end{array}\right\}$

- On obtient ces triplets par la règle d'affectation
- On les combine grâce à la règle de séquence

# Règle d'affectation

Chacun de ces triplets, par exemple $\left\{ \begin{array}{l} t = \alpha \\ a = \beta \end{array} \right\} b := t \left\{ \begin{array}{l} b = \alpha \\ a = \beta \end{array} \right\}$ est obtenu par la règle d'affectation.

## Règle d'affectation

Pour toute formule $P$, variable $x$ et expression $e$, on peut utiliser l'axiome $\{P[e/x]\}\, x := e\, \{P\}$, ce qu'on note

$$\frac{}{\{P[e/x]\}\, x := e\, \{P\}}$$

## Exemples

- $\{e + 3 > 0\}\, x := e\, \{x + 3 > 0\}$
- $\{x + 1 > 0\}\, x := x + 1\, \{x > 0\}$

# Règle de la séquence

- On a obtenu des triplets

$$\{P_1\}\, t := a\, \{P_2\} \quad \{P_2\}\, a := b\, \{P_3\} \quad \{P_3\}\, b := t\, \{P_4\}$$

- On peut en déduire $\{P_1\}\, t := a;\ a := b;\ b := t\, \{P_4\}$ par la règle de la séquence.

## Règle de la séquence

Des triplets $\{P\}\, i_1\, \{Q\}$ et $\{Q\}\, i_2\, \{R\}$, on peut déduire le triplet $\{P\}\, i_1; i_2\, \{R\}$, ce qu'on note par la règle :

$$\frac{\{P\}\, i_1\, \{Q\} \quad \{Q\}\, i_2\, \{R\}}{\{P\}\, i_1; i_2\, \{R\}}$$

# Conditionnelle

Montrons

$$\{-2 \leq x \leq -2\} \text{ if } x < 0 \text{ then } y := x+3 \text{ else } y := 5-x \{1 \leq y \leq 5\}$$

## Règle de la conditionnelle

$$\frac{\{b \wedge P\}\, i_1\, \{Q\} \quad \{\neg b \wedge P\}\, i_2\, \{Q\}}{\{P\} \text{ if } b \text{ then } i_1 \text{ else } i_2 \{Q\}}$$

Il suffit de montrer

- $\{(-2 \leq x < 2) \wedge (x < 0)\}\, y := x + 3\, \{1 \leq y \leq 5\}$
- et $\{(-2 \leq x \leq 2) \wedge \neg(x < 0)\}\, y := 5 - x\, \{1 \leq y \leq 5\}$

# Conditionnelle (suite)

Il suffit donc de montrer

- $\{(-2 \leq x < 2) \wedge (x < 0)\}\, y := x + 3\, \{1 \leq y \leq 5\}$
- et $\{(-2 \leq x \leq 2) \wedge \neg(x < 0)\}\, y := 5 - x\, \{1 \leq y \leq 5\}$

Avec la règle de la séquence, on obtient respectivement :

- $\{1 \leq x + 3 \leq 5\}\, y := x + 3\, \{1 \leq y \leq 5\}$
- $\{1 \leq 5 - x \leq 5\}\, y := 5 - x\, \{1 \leq y \leq 5\}$

Besoin d'une nouvelle règle : la règle d'affaiblissement.

# Affaiblissement

## Règle d'affaiblissement

$$\frac{\{P'\}\, i\, \{Q'\} \quad P \Rightarrow P' \quad Q' \Rightarrow Q}{\{P\}\, i\, \{Q\}}$$

Ici, on a bien :

- $[(-2 \leq x < 2) \wedge (x < 0)] \Rightarrow (1 \leq x + 3 \leq 5)$
- $[(-2 \leq x \leq 2) \wedge \neg(x < 0)] \Rightarrow (1 \leq 5 - x \leq 5)$

# Boucle

Montrons

$$\{n = 15\}\,\text{while}\,n > 0\ \ \text{do}\,n := n - 3\,\{n = 0\}$$

## Règle de la boucle

$$\frac{\{b \wedge P\}\,i\,\{P\}}{\{P\}\,\text{while}\,b\ \ \text{do}\,i\,\{\neg b \wedge P\}}$$

($P$ appelé invariant de la boucle)

Attention :

- $\{n = 15 \wedge n > 0\}\,n := n - 3\,\{n = 15\}$ n'est pas valide.
- $\{n \geq 0 \wedge n > 0\}\,n := n - 3\,\{n \geq 0\}$ non plus.
- Trouver un invariant n'est pas nécessairement facile.

# Un invariant qui convient

Invariant de la boucle :

- $n \geq 0$
- et $n$ multiple de $3$.

En effet, on a successivement

$$\{n \in 3\mathbb{N} \wedge n > 0\}\, n := n - 3\, \{n \in 3\mathbb{N}\} \qquad \text{affectation}$$

$$\{n \in 3\mathbb{N}\}\, \text{while } n > 0 \text{ do } n := n - 3\, \{n \leq 0 \wedge n \in 3\mathbb{N}\} \qquad \text{boucle}$$

$$\{n = 15\}\, \text{while } n > 0 \text{ do } n := n - 3\, \{n = 0\} \qquad \text{affaiblissement}$$

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Correction

## Théorème

*Les règles données permettent uniquement de dériver des triplets valides.*

# Complétude

## Théorème

*Les règles données sont suffisantes pour dériver tous les triplets valides.*

# Plus faible précondition

## Théorème

Pour tout programme $i$, et toute postcondition $Q$, il existe une plus faible précondition $P$ telle que $\{P\}\, i\, \{Q\}$ soit valide.
Elle est notée $\mathrm{wp}(i, Q)$. On a :

1. $\{\mathrm{wp}(i, Q)\}\, i\, \{Q\}$ valide
2. et pour toute proposition $P'$, $\{P'\}\, i\, \{Q\}$ est valide si et seulement si $P' \Rightarrow \mathrm{wp}(i, Q)$

De plus, on peut calculer $\mathrm{wp}(i, Q)$ de façon effective.

## Remarque

Permet théoriquement de ramener mécaniquement la preuve de programme à celle d'une implication mathématique.

# En pratique

Les préconditions calculées sont naturelles, sauf pour les boucles :

- $\mathrm{wp}(\text{while } e \text{ do } i, Q)$ est presque la paraphrase

  *si on exécute* while $e$ do $i$, *alors on aura* $Q$

- aucun intérêt pour démontrer la correction du programme.

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Programmes annotés

On annote les boucles explicitement avec des invariants :
On n'écrit plus

$$\text{while } e \text{ do } i$$

mais

$$\text{while } e \text{ invariant } P \text{ do } i$$

## Règle de la boucle

$$\frac{\{b \wedge P\}\, i\, \{P\}}{\{P\}\, \text{while } b \text{ invariant } P \text{ do } i\, \{\neg b \wedge P\}}$$

# Correction partielle / correction totale

Vrai ou faux ?

$\{x > 0\}\,\mathsf{while}\,x > 0\,\mathsf{invariant}\ x > 0\ \mathsf{do}\,x := x\,\{\neg(x > 0) \wedge (x > 0)\}$

# Correction partielle / correction totale

Vrai ou faux ?

$\{x > 0\}\,\text{while}\,x > 0\,\text{invariant}\ x > 0\ \text{do}\,x := x\,\{\neg(x > 0) \wedge (x > 0)\}$

- C'est un triplet de Hoare valide

# Correction partielle / correction totale

Vrai ou faux ?

$\{x > 0\}\,\text{while}\,x > 0\,\text{invariant}\ x > 0\ \text{do}\,x := x\,\{\neg(x > 0) \wedge (x > 0)\}$

- C'est un triplet de Hoare valide
- On n'a modélisé que la correction *partielle*.

# Correction partielle / correction totale

Vrai ou faux ?

$\{x > 0\}\,\text{while}\,x > 0\,\text{invariant}\ x > 0\ \text{do}\,x := x\,\{\neg(x > 0) \wedge (x > 0)\}$

- C'est un triplet de Hoare valide
- On n'a modélisé que la correction *partielle*.
- On aimerait modéliser la correction *totale*.

# Correction partielle / correction totale

Vrai ou faux ?

$\{x > 0\}\,\text{while}\,x > 0\,\text{invariant}\ x > 0\ \text{do}\,x := x\,\{\neg(x > 0) \wedge (x > 0)\}$

- C'est un triplet de Hoare valide
- On n'a modélisé que la correction *partielle*.
- On aimerait modéliser la correction *totale*.
- Correction totale = correction partielle + terminaison

# Correction totale

Jusqu'ici $\{P\}\, i\, \{Q\}$ signifiait

> *Pour toute valeur initiale des variables vérifiant $P$ telle que l'exécution de $i$ termine, la valeur des variables à la fin de l'exécution vérifie $Q$.*

À partir de maintenant $\{P\}\, i\, \{Q\}$ signifiera

> *Pour toute valeur initiale des variables vérifiant $P$, l'exécution de $i$ termine et la valeur des variables à la fin de l'exécution vérifie $Q$.*

Une seule règle de raisonnement à changer : celle de la boucle.

# Notion de variant

On annote les boucles :

- avec un invariant pour montrer la correction partielle

# Notion de variant

On annote les boucles :

- avec un invariant pour montrer la correction partielle
- plus un variant pour montrer la terminaison

# Notion de variant

On annote les boucles :

- avec un invariant pour montrer la correction partielle
- plus un variant pour montrer la terminaison

# Notion de variant

On annote les boucles :

- avec un invariant pour montrer la correction partielle
- plus un variant pour montrer la terminaison

## Variant

Un variant est une quantité entière positive qui décroît strictement à chaque tour de boucle.
Syntaxe :

$$\text{while } e_1 \text{ invariant } P \text{ variant } e_2 \text{ do } i$$

# Règle de la boucle

## Règle de la boucle

$$\frac{\{b \wedge P \wedge e \in \mathbb{N} \wedge e = \eta\}\, i\, \{P \wedge e \in \mathbb{N} \wedge e < \eta\}}{\{P\}\, \text{while}\, b\ \ \text{invariant}\ \ P\, \text{variant}\ \ e\ \ \text{do}\, i\, \{\neg b \wedge P\}}$$

($\eta$ variable n'apparaissant pas dans $i$)

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Le programme

Montrer que le programme Python suivant est correct :

```
# précondition: a = α ∈ ℤ et b = β ∈ ℕ.
r = 1
while b > 0:
  if b % 2 == 1:
    b = b - 1
    r = r * a
  else:
    b = b // 2
    a = a * a
# postcondition r = α^β
```

# Le programme annoté

```
# précondition: a = α ∈ ℤ et b = β ∈ ℕ.
r = 1
while b > 0:
  # invariants: a ∈ ℤ, b ∈ ℕ et r × a^b = α^β.
  # variant: b
  if b % 2 == 1:
    b = b - 1
    r = r * a
  else: # b est pair
    b = b // 2
    a = a * a
# postcondition r = α^β
```

# Correction de l'invariant

Montrons que les invariants donnés dans le programme sont corrects :

- Ils sont vérifiés avant le premier tour de boucle. En effet, on a alors $a = \alpha \in \mathbb{Z}$, $b = \beta \in \mathbb{N}$ et $r \times a^b = 1 \times \alpha^\beta = \alpha^\beta$.
- Supposons qu'au début d'un tour de boucle, la condition de boucle et les invariants soient vrais, montrons alors que les invariants sont vrais à la fin du tour. Notons $v_a$, $v_b$ et $v_r$ les valeurs respectives de $a$, $b$ et $r$ au début du tour. On a $v_b > 0$ et $v_a \in \mathbb{Z}$, $v_b \in \mathbb{N}$ et $v_r v_a^{v_b} = \alpha^\beta$ (invariant). Alors :
  - Si $v_b$ est impair, à la fin du tour, on a $b = v_b - 1 \in \mathbb{N}$, $a = v_a$ et $r = v_r v_a$, d'où $r \times a^b = v_r v_a v_a^{v_b - 1} = v_r v_a^{v_b} = \alpha^\beta$.
  - Si $v_b$ est pair, à la fin du tour, on a $b = v_b/2 \in \mathbb{N}$, $a = v_a^2 \in \mathbb{Z}$, et $r \times a^b = v_r (v_a^2)^{v_b/2} = v_r v_a^{v_b} = \alpha^\beta$.

Donc les invariants donnés dans le programme sont corrects.

# Décroissance stricte du variant

On a vu que le variant était un entier naturel. Montrons que la valeur de $b$ décroît strictement à chaque tour de boucle. Notons $v_b$ sa valeur au début d'un tour de boucle. On sait qu'on a $v_b \in \mathbb{N}$ et $v_b > 0$.

- Si $v_b$ est impair, alors à la fin de la boucle, $b = v_b - 1 < v_b$.
- Si $v_b$ est pair, alors à la fin de la boucle, $b = v_b/2 < v_b$ car $v_b > 0$.

Donc le variant décroît strictement à chaque tour de boucle.
Donc la boucle termine.

# Correction du programme

- La boucle termine donc le programme termine.
- À la fin de la boucle, on a $b \leq 0$ (condition de boucle) ainsi que $b \in \mathbb{N}$ et $r \times a^b = \alpha^\beta$ (invariant). D'où $b = 0$ et $r = \alpha^\beta$.

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Programme

Montrer que le programme Python suivant est correct :

```
# précondition: t tableau non vide.
n = len(t)
k = 0
for i in range(1, n):
  if t[i] > t[k]:
    k = i
# postcondition t[k] est le maximum du tableau.
```

# Un for est un while déguisé :

## Boucle définie

```
for i in range(a, b):
    ...
```

## Boucle indéfinie équivalente

```
i = a
while i < b:
    ...
    i = i + 1
i = i - 1
```

(si $a$ et $b$ inchangées durant la boucle)

# Règle pour le for

Si les valeurs de $a$ et $b$ sont inchangées durant la boucle et si

- $I[a/i]$ vrai avant l'entrée dans la boucle `for`
- et, pour chaque exécution du corps de la boucle, $I$ vrai avant l'exécution implique $I[i+1/i]$ vrai après l'exécution.

Alors $I[b/i]$ est vrai à la fin de la boucle.

## Règle

Si les valeurs de $a$ et $b$ sont inchangées durant la boucle

$$\frac{\{P\}\ instr\ \{P[i+1/i]\}}{\{P[a/i]\}\ \text{for}\ i\ \text{in}\ \text{range}(a,b):\ \ instr\ \{P[b/i]\}}$$

# Programme annoté

```
# précondition: t tableau non vide.
n = len(t)
k = 0
for i in range(1, n):
  # t[k] est le maximum de t[0:i]
  if t[i] > t[k]:
    k = i
# postcondition t[k] est le maximum du tableau.
```

# Lignes directrices

1 Introduction
- Motivation : correction des programmes
- Méthodes formelles

2 Logique de Hoare
- Cadre
- Règles
- Résultats théoriques
- Mécanisation de la preuve de programmes

3 Exemples
- Exponentiation rapide
- Recherche du maximum d'un tableau
- Recherche dichotomique

4 Conclusion

# Contexte

Étant donné :

- un tableau $t$ de $n$ valeurs triées par ordre croissant
- une valeur $v$

Retourner $k$ tel que $t[k] = v$ ($-1$ si $t$ ne contient pas $v$), en $O(\log n)$ comparaisons.

- Algorithme fondamental présent dans les cursus informatique depuis longtemps.

# Contexte

Étant donné :

- un tableau $t$ de $n$ valeurs triées par ordre croissant
- une valeur $v$

Retourner $k$ tel que $t[k] = v$ ($-1$ si $t$ ne contient pas $v$), en $O(\log n)$ comparaisons.

- Algorithme fondamental présent dans les cursus informatique depuis longtemps.
- Publié pour la première fois en 1946.

# Contexte

Étant donné :

- un tableau $t$ de $n$ valeurs triées par ordre croissant
- une valeur $v$

Retourner $k$ tel que $t[k] = v$ ($-1$ si $t$ ne contient pas $v$), en $O(\log n)$ comparaisons.

- Algorithme fondamental présent dans les cursus informatique depuis longtemps.
- Publié pour la première fois en 1946.
- Publié pour la première fois sans bug en 1962.

# Contexte

Étant donné :

- un tableau $t$ de $n$ valeurs triées par ordre croissant
- une valeur $v$

Retourner $k$ tel que $t[k] = v$ ($-1$ si $t$ ne contient pas $v$), en $O(\log n)$ comparaisons.

- Algorithme fondamental présent dans les cursus informatique depuis longtemps.
- Publié pour la première fois en 1946.
- Publié pour la première fois sans bug en 1962.
- En 1986 : 90% des programmeurs incapables de l'écrire sans bug en 2h (expérience de Bentley).

# Un programme buggué

```
n = len(t)
i = 0
j = n-1
while j - i >= 1:
  m = (i+j) // 2
  if v <= t[m]:
     j = m-1
  else:
     i = m
if t[i] == v: return i
else: return -1
```

# Programme annoté

```
n = len(t)
i = 0
j = n-1
while j - i >= 1:
    # invariant : 0 ≤ i ≤ j < n et v ∈ t ⇒ v ∈ {t[i], ..., t[j]}
    # variant : j − i
    m = (i+j) // 2 # i ≤ m ≤ j
    if v <= t[m]: # alors v ∈ t ⇒ v ∈ {t[i], ..., t[m]}
        j = m-1
    else: # v > t[m] donc v ∈ t ⇒ v ∈ {t[m+1], ..., t[j]}
        i = m
if t[i] == v: return i
else: return -1
```

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$

Invariant :

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$
- c'est faux !

Invariant :

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$
- c'est faux !
- le programme peut boucler (si $i = 0$ et $j = 1$)

Invariant :

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$
- c'est faux !
- le programme peut boucler (si $i = 0$ et $j = 1$)

Invariant :

- on doit montrer que l'invariant est vrai avant l'entrée dans la boucle.

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$
- c'est faux !
- le programme peut boucler (si $i = 0$ et $j = 1$)

Invariant :

- on doit montrer que l'invariant est vrai avant l'entrée dans la boucle.
- en particulier, on doit montrer $0 \leq i \leq j < n$.

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$
- c'est faux !
- le programme peut boucler (si $i = 0$ et $j = 1$)

Invariant :

- on doit montrer que l'invariant est vrai avant l'entrée dans la boucle.
- en particulier, on doit montrer $0 \leq i \leq j < n$.
- c'est faux si $n = 0$ !

# Variant

- on doit montrer $j - i \geq 1 \Rightarrow j - \left\lfloor \frac{i+j}{2} \right\rfloor < j - i$
- c'est faux !
- le programme peut boucler (si $i = 0$ et $j = 1$)

Invariant :

- on doit montrer que l'invariant est vrai avant l'entrée dans la boucle.
- en particulier, on doit montrer $0 \leq i \leq j < n$.
- c'est faux si $n = 0$ !
- le programme plante si $t$ est vide

# Corriger les bugs

Deux approches possibles

- Corriger les bugs et revérifier la correction.

# Corriger les bugs

Deux approches possibles

- Corriger les bugs et revérifier la correction.
- Reprendre la conception du programme :

# Corriger les bugs

Deux approches possibles

- Corriger les bugs et revérifier la correction.
- Reprendre la conception du programme :
  - On jette le premier programme à la poubelle

# Corriger les bugs

Deux approches possibles

- Corriger les bugs et revérifier la correction.
- Reprendre la conception du programme :
  - On jette le premier programme à la poubelle
  - On réfléchit dès le départ à l'invariant et au variant

# Corriger les bugs

Deux approches possibles

- Corriger les bugs et revérifier la correction.
- Reprendre la conception du programme :
  - On jette le premier programme à la poubelle
  - On réfléchit dès le départ à l'invariant et au variant
  - On écrit le détail du code seulement ensuite

# Ce qu'on veut

- Chercher une valeur $v$ par dichotomie dans un tableau $t$.

# Ce qu'on veut

- Chercher une valeur $v$ par dichotomie dans un tableau $t$.
- Variant naturel : taille $n$ du sous-tableau dans lequel on cherche.

# Ce qu'on veut

- Chercher une valeur $v$ par dichotomie dans un tableau $t$.
- Variant naturel : taille $n$ du sous-tableau dans lequel on cherche.
- On coupe ce sous-tableau en deux sous-tableaux de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ et on cherche dans l'un des deux.

# Ce qu'on veut

- Chercher une valeur $v$ par dichotomie dans un tableau $t$.
- Variant naturel : taille $n$ du sous-tableau dans lequel on cherche.
- On coupe ce sous-tableau en deux sous-tableaux de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ et on cherche dans l'un des deux.
- Pour que ce nouveau sous-tableau soit de taille strictement plus petite, il suffit d'avoir $\lceil n/2 \rceil < n$, i.e. $n \geq 2$.

# Ce qu'on veut

- Chercher une valeur $v$ par dichotomie dans un tableau $t$.
- Variant naturel : taille $n$ du sous-tableau dans lequel on cherche.
- On coupe ce sous-tableau en deux sous-tableaux de tailles $\lfloor n/2 \rfloor$ et $\lceil n/2 \rceil$ et on cherche dans l'un des deux.
- Pour que ce nouveau sous-tableau soit de taille strictement plus petite, il suffit d'avoir $\lceil n/2 \rceil < n$, i.e. $n \geq 2$.
- Invariant : si $v \in t$, alors $v$ est dans le sous-tableau de $t$ de taille $n$ commençant à un certain indice $i$ :

$$v \in t \Rightarrow v \in t[i : i + n]$$

# Squelette du programme

```
...
while n >= 2:
    # invariant : n ≥ 0 et ⟦i, i + n⟦ ⊂ ⟦0, |t|⟦
    # et v ∈ t ⇒ v ∈ t[i : i + n]
    # variant : n
    k = n // 2
    if ...: # alors v ∈ t ⇒ v ∈ t[i : i + k]
        n = k
    else: # v ∈ t ⇒ v ∈ t[i + k : i + n]
        i = i+k
        n = n - k
# n < 2 et n ≥ 0
...
```

# Le programme

```
n, i = len(t), 0
while n >= 2:
    # invariant : n ≥ 0 et ⟦i, i + n⟦ ⊂ ⟦0, |t|⟦
    # et v ∈ t ⇒ v ∈ t[i : i + n]
    # variant : n
    k = n // 2
    if m < t[i+k]: # alors v ∈ t ⇒ v ∈ t[i : i + k]
        n = k
    else: # v ∈ t ⇒ v ∈ t[i + k : i + n]
        i = i+k
        n = n - k
# n < 2 et n ≥ 0 et v ∈ t ⇒ v ∈ t[i : i + n]
if n == 1 and t[i] == m: return i
else: return -1
```

# Conclusion

La sémantique axiomatique permet :

- de spécifier des programmes impératifs
- d'en montrer la terminaison et la correction
- de construire des programmes corrects du premier coup