Algorithmique procédurale

Les invariants

Équipe pédagogique

CY Tech

Bibliographie - Sitographie

- Houcine Senoussi
- Aurélie Lagoutte :
  https://extradoc.univ-nantes.fr/pluginfile.php/256766/mod_resource/content/4/1%C3%A8re--Algorithmique--Cours--Preuve.pdf

Rappels - Objectifs

- L'objectif est d'apprendre à analyser les algorithmes.
- Analyser un algorithme s'effectue en trois étapes :
  1. **Terminaison** : prouver qu'il termine.
  2. **Correction** : prouver qu'il résout le problème.
  3. **Complexité** : évaluer son coût en temps et en espace.

Pourquoi analyser un algorithme ?

préconditions : une taille n.
postconditions : renvoie le résultat de la somme des entiers de 1 à n.

fonction somme_entiers(n: entier): entier
Variables
  i,s: entier
début
  s $\leftarrow$ 1
  pour i $\leftarrow$ 1 **à** n **faire**
    s $\leftarrow$ s+i
  fin pour
  retourner s
fin fonction

- L'algorithme renvoie la somme des entiers de 1 à n en additionnant à chaque étape i à la somme qu'on a calculée précédemment

Pourquoi analyser un algorithme ?

préconditions : une taille n.
postconditions : renvoie le résultat de la somme des entiers de 1 à n.

fonction somme_entiers(n: entier): entier
Variables
  i,s: entier
début
  s ← 1
  pour i ← 1 **à** n **faire**
    s ← s+i
  fin pour
  retourner s
fin
fin fonction

- NON !!! Cet algorithme renvoie 2 pour l'appel somme_entiers(1)

Pourquoi analyser un algorithme ?

- Pour vérifier et convaincre que notre algorithme répond au problème posé.
  1. Vérifier la correction d'un algorithme, c'est vérifier qu'il renvoie la réponse correcte
  2. Vérifier la terminaison d'un algorithme, c'est vérifier que l'algorithme se termine

(éventuellement en précisant au bout de combien de "temps" / itérations / opérations)

Terminaison et correction

- Pour la plupart des (suite d') instructions, l'étude de la terminaison et de la correction des algorithmes ne présente pas de difficultés.
- Deux types de suites d'instructions doivent être traités à part :
  - les boucles : qui sont étudiées à l'aide des **invariants de boucles**. Il s'agit d'une démarche semblable à la récurrence mathématique.
  - les appels récursifs : ce problème sera étudié dans un autre cours.

Invariant de boucle

- Généralement, on se concentre sur l'analyse des boucles (**Pour** et **Tant Que**).
- On cherche à écrire une phrase qui reste vraie pendant tout le déroulement de la boucle.
- Vocabulaire : on appelle cette "phrase" une propriété ou encore un invariant de boucle.

Invariant de boucle

- Un invariant de boucle est une assertion vérifiant les trois propriétés suivantes :
  - **Initialisation** : elle est vraie avant la première itération de la boucle.
  - **Conservation** (transmission) : si elle est vraie avant une itération de la boucle, alors elle est vraie avant la suivante.
  - **Terminaison** : à la fin de la boucle, elle fournit une propriété permettant de montrer la correction de l'algorithme.

Invariant de boucle

préconditions : une taille n.
postconditions : renvoie le résultat de la somme des entiers de 1 à n.

```
fonction somme_entiers_correcte(n: entier): entier
Variables
   i,s: entier
début
   s ← 0
   pour i ← 1 à n faire
      s ← s+i
   /* Invariant de boucle : à cette étape de l'algorithme, s contient
      la somme des entiers de 1 à i (inclus) */
   fin pour
   retourner s
fin
fin fonction
```

Invariant de boucle

préconditions : une taille n.
postconditions : renvoie le résultat de la somme des entiers de 1 à n.

```
fonction somme_entiers_correcte(n: entier): entier
Variables
   i,s: entier
début
   s ← 0
   pour i ← 1 à n faire
      s ← s+i
   /* Itération 1 : on a bien i à 1 et s à 1 */
   /* Invariant de boucle : à cette étape de l'algorithme, s contient
      la somme des entiers de 1 à i (inclus) */
   fin pour
   retourner s
fin
fin fonction
```

Invariant de boucle

```
fonction somme_entiers_correcte(n: entier): entier
D'une itération à l'autre : retraduisons l'invariant de boucle aux
   autres endroits utiles pour l'analyse
Variables
   i,s: entier
début
   s ← 0
   pour i ← 1 à n faire
   /* s somme entiers de 1 à i-1 (invariant de l'itération précédente)
      */
      s ← s+i
   /* Invariant de boucle : à cette étape de l'algorithme, s contient
      la somme des entiers de 1 à i (inclus) */
   fin pour
   retourner s
fin
fin fonction
```

Invariant de boucle

```
fonction somme_entiers_correcte(n: entier): entier
On vérifie la cohérence de nos propriétés à tous les endroits où on
   les a explicités OK
Variables
   i,s: entier
début
   s ← 0
   pour i ← 1 à n faire
   /* s somme entiers de 1 à i-1 (invariant de l'itération précédente)
      */
      s ← s+i
   /* Invariant de boucle : à cette étape de l'algorithme, s contient
      la somme des entiers de 1 à i (inclus) */
   fin pour
   retourner s
fin
fin fonction
```

Invariant de boucle

fonction somme_entiers_correcte(n: entier): entier
Variables
i,s: entier
début
s ← 0
pour i ← 1 **à** n **faire**
/ s somme entiers de 1 à i-1 (invariant de l'itération précédente) */*
s ← s+i
/ Invariant de boucle : à cette étape de l'algorithme, s contient la somme des entiers de 1 à i (inclus) */*
Ici, juste avant de sortir de la dernière itération de boucle : i vaut n. Donc, grâce à l'invariant, s vaut 1+ ... +n Correction algo OK
fin pour
retourner s
fin
fin fonction

Invariant de boucle

- Attention, il faut bien préciser à quel endroit vous placez votre invariant de boucle.

Invariant de boucle

```
fonction somme_entiers_correcte(n: entier): entier
Variables
   i,s: entier
début
   s ← 0
   pour i ← 1 à n faire
   /* Invariant de boucle : à cette étape de l'algorithme, s contient
      la somme des entiers de 1 à i (inclus) - FAUX : par exemple
      à la première itération i vaut 1 et s vant 0 */
      s ← s+i
   fin pour
   retourner s
fin
fin fonction
```

Invariant de boucle

- Attention, il faut bien préciser à quel endroit vous placez votre invariant de boucle.
- Il faut bien vérifier qu'il est vrai à la première itération (comme le cas de base d'une récurrence)

Invariant de boucle

```
fonction somme_entiers_fausse(n: entier): entier
Variables
   i,s: entier
début
   s ← 1
   pour i ← 1 à n faire
      s ← s+i
   /* Invariant de boucle : à cette étape de l'algorithme, s contient
      la somme des entiers de 1 à i (inclus) - FAUX : par exemple
      à la première itération i vaut 1 et s vant 2 */
   fin pour
   retourner s
fin
fin fonction
```

Invariant de boucle

- Attention, il faut bien préciser à quel endroit vous placez votre invariant de boucle.
- Il faut bien vérifier qu'il est vrai à la première itération (comme le cas de base d'une récurrence)
- Il faut vérifier qu'il est vrai d'une itération à l'autre

Invariant de boucle

- Attention, il faut bien préciser à quel endroit vous placez votre invariant de boucle.
- Il faut bien vérifier qu'il est vrai à la première itération (comme le cas de base d'une récurrence)
- Il faut vérifier qu'il est vrai d'une itération à l'autre
- Il faut généralement porter un soin particulier à la dernière itération

Invariant de boucle

- Attention, il faut bien préciser à quel endroit vous placez votre invariant de boucle.
- Il faut bien vérifier qu'il est vrai à la première itération (comme le cas de base d'une récurrence)
- Il faut vérifier qu'il est vrai d'une itération à l'autre
- Il faut généralement porter un soin particulier à la dernière itération
- Une fois l'invariant prouvé, il faut vérifier que l'algorithme fait ce qui est demandé dans l'énoncé grâce à cet invariant.

Terminaison d'un algorithme

- Il est important de vérifier que l'algorithme se termine

i $\leftarrow$ 0
tant que (i $<$ 10) **faire**
  écrire(i)
fin tant que
/ Boucle infinie car i n'est jamais modifié */*

Terminaison d'un algorithme

- Parmi les instructions que l'on a vue jusqu'à présent : seule une instruction **Tant que** peut mener à un algorithme qui ne termine pas.
- On limite donc l'analyse de terminaisons aux boucles **Tant que**.

Terminaison d'un algorithme

- Pour prouver qu'une boucle **Tant que** se termine : il faut prouver qu'à chaque étape, on se rapproche "d'au moins un cran" d'un cas d'arrêt.

i $\leftarrow$ 0
tant que (i $<$ 10) **faire**
/ Cas d'arrêt : i supérieur ou égal à 10 */*
   écrire(i)
   i $\leftarrow$ i+1
/ i = 0 et est incrémenté de 1 à chaque étape : 0,1,2,......,10* **OK**
   /
fin tant que

Terminaison d'un algorithme

- Pour prouver qu'une boucle **Tant que** se termine : il faut prouver qu'à chaque étape, on se rapproche "d'au moins un cran" d'un cas d'arrêt.

i $\leftarrow$ 0
tant que (i $<$ 10) **faire**
/ Cas d'arrêt : i supérieur ou égal à 10 */*
  écrire(i)
  i $\leftarrow$ i-1
/ i = 0 et est décrémenté de 1 à chaque étape : 0,-1,-2,...* **jamais** $>$ **10** */
fin tant que

Exemple : Tri par insertion

préconditions : un tableau d'entiers tab de taille N.
postconditions : le même tableau trié.

```
pour taille ← 2 à N faire
    c ← tab[taille]
    i ← taille-1
    tant que (i>0) et (tab[i]>c) faire
        tab[i+1] ← tab[i]
        i ← i-1
    fin tant que
    tab[i+1] ← c
fin pour
```

Exemple : Tri par insertion

- Idée de l'algorithme : à la fin de l'étape correspondant à taille=t, les t éléments le plus à gauche sont triés.
- Nous traduisons cette idée en un invariant de boucle défini comme suit :

Invariant de boucle du tri par insertion

Le sous tableau tab[1..t-1] se compose des éléments :

1. qui occupaient les $t - 1$ premières positions du tableau initial,
2. et qui ont été triés.

Exemple : Tri par insertion

- **Initialisation** : La première itération correspond à $t = 2$. Au début de cette itération, le sous-tableau tab[1..t-1]=tab[1] contient bien le premier élément du tableau.
- **Conservation** : Supposons que la propriété soit vraie jusqu'à l'itération $t - 1$. Le corps de l'itération t consiste à :
  - Trouver **la** valeur de k telle que $k = max\{h \leq t - 1 \; ET \; tab[h] \leq tab[t]\}$.
  - Décaler $tab[k + 1]$, ..., $tab[t - 1]$ d'un indice vers la droite.
  - Affecter à $tab[k + 1]$ la valeur $tab[t]$.

  Les valeurs présentes dans tab[1..t] sont donc bien celles de tab[1..t] au début de la boucle et elles sont triées. L'assertion est donc vraie à la fin de l'itération t.

Exemple : Tri par insertion

- **Terminaison** : La condition qui force la boucle à s'arrêter est $t > N$. Plus exactement $t = N + 1$ puisque le pas de la boucle est égal à 1. Pour cette valeur de t l'invariant de boucle est "le sous-tableau $tab[1..N]$ est composé des éléments initiaux de $tab[1..N]$ et ils ont été triés". Cela signifie simplement que le tableau a été trié, ce qui prouve l'algorithme.

Exercices

- Trouvez un invariant de boucle pour le calcul (itératif) de la factorielle.
- Idem pour le calcul de la division euclidienne par la méthode des soustractions.