UCLouvain

Algorithmique

Cours préparatoires au master en sciences informatiques

Vincent Branders

2022–2023

Preuves par induction et invariants

Preuve par induction

Objectifs

Prouver l'exactitude d'un algorithme, en particulier récursif

Plus généralement, prouver l'exactitude d'un énoncé

Un premier exemple de preuve par induction


```
...
for i ← 1 to n do
    ...
    for j ← 1 to i do
        ...
    ...
...
```

Nombre d'itérations :

$$\sum_{i=1}^{n} i = 1 + \cdots + n = \frac{n(n+1)}{2}$$

Preuve

1. **Cas de base** : vrai pour $n = 1$ (ou $n = 0$)

   $\sum_{i=1}^{1} i = \frac{1(1+1)}{2} = \frac{2}{2} = 1$

2. **Induction** :
   - **Si** le résultat est **vrai pour** $n-1$,
   - **alors** il est **vrai pour** n

   $\sum_{i=1}^{n} i = \left(\sum_{i=1}^{n-1} i\right) + n \overset{\text{Hyp.}}{=} \frac{(n-1)n}{2} + n = \frac{n^2}{2} - \frac{n}{2} + n = \frac{n^2}{2} + \frac{n}{2} = \frac{n(n+1)}{2}$

3. **Conclusion** : le résultat est vrai pour tout $n \geq 1$

Récursion versus Induction

Algorithm: fact
Input: Un entier n
Output: $n!$
if $n == 0$ **then**
  return 1
return $n \times$ `fact(n-1)`

Preuve de l'exactitude par induction

1. Cas de base : `fact(0)` $= 1 = 0!$
2. Induction :
   - Si `fact(n-1)` est correct
   - alors `fact(n)` est correct

   `fact(n)` $= n \times$ `fact(n-1)` $\overset{\text{Hyp.}}{=} n \times (n-1)! = n!$
3. Conclusion : `fact(n)` calcule correctement $n!$ pour tout $n \geq 0$

- L'appel récursif est une réduction à un sous-problème
- Le raisonnement inductif est une généralisation à un problème plus large
  - Cas de base pour une (ou plusieurs) valeur(s) n_0 (souvent 0 ou 1)
  - Généralisation pour tout $n \geq n_0$

Prouvez par induction :

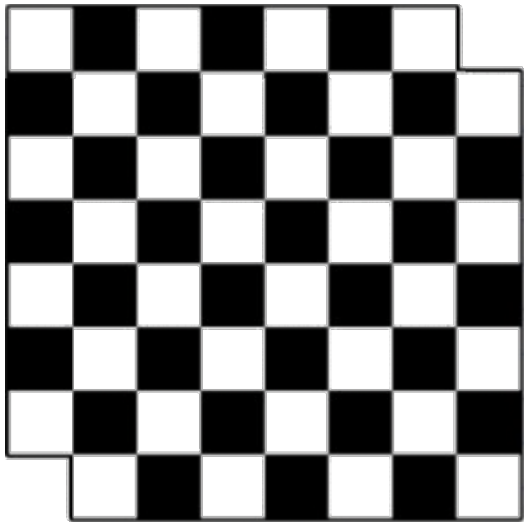
1. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
2. $\sum_{i=1}^{n} a^i = \frac{a(a^n-1)}{a-1} \quad \forall 0 < a \neq 1$
3. https://inginious.org/course/prepa-master/Algo_22_rec_vs_ind

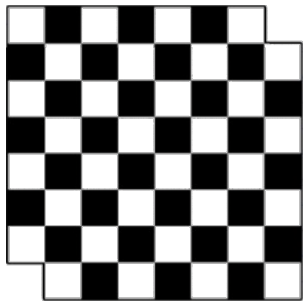
Un petit problème

Question

Comment couvrir l'échiquier avec des dominos ?

- les 64 - 2 = 62 cases doivent être couvertes
- chaque domino couvre exactement 2 cases, horizontalement ou verticalement
- les dominos ne peuvent pas se chevaucher ni déborder de l'échiquier





- Chaque domino couvrant exactement 2 cases, nous avons besoin de 31 dominos pour couvrir 62 cases

Invariant

Chaque domino couvre 2 cases adjacentes qui sont forcément de couleur opposée

- Lorsque 30 dominos auront été posés, il restera 2 cases à couvrir
  - Ces 2 cases sont nécessairement toutes les deux de couleur blanche car 2 cases noires ont été supprimées de l'échiquier
  - Le dernier domino ne peut pas couvrir 2 cases de couleur identique
  - Il n'y a pas de solution !

Invariants

Définition

- Un invariant est une propriété qui ne varie pas (p.ex. toujours vraie)
  - Exemple : chaque domino couvre 2 cases adjacentes qui sont forcément de couleur opposée, quelque soit la manière dont sont disposés les dominos sur l'échiquier
- Les invariants sont utiles pour démontrer
  - qu'il y a ou non une solution à un problème
  - qu'un algorithme est exact

Invariant de boucle

Définition

Un invariant de boucle est une propriété qui est vraie

- lors de l'initialisation de la boucle
- après chaque itération de la boucle
- après terminaison de la boucle

Reformulation d'une boucle **for** en **while**


```
for i ← 1 to n − 1 do
    ...
...
```

```
i ← 1                                   // Initialisation
while i ≤ n − 1 do
    ...
    i ← i + 1                           // Itération
                                        // Terminaison (i == n)
...
```

Tri par insertion

Algorithm: InsertionSort
Input: Un tableau A de n éléments
Output: Le tableau A trié en ordre croissant

```
i ← 1                                                    // Initialisation
while i ≤ n − 1 do
    key ← A[i]
    j ← i − 1
    // Insertion de A[i] dans le tableau trié A[0 . . . i1]
    while j ≥ 0 and A[j] > key do
        A[j + 1] ← A[j] // Déplacement d'une position
        j ← j − 1
    A[j + 1] ← key
    i ← i + 1                                            // Itération
return A                                                 // Terminaison
```

Invariant du tri par insertion

Le tableau $A[0, \ldots, i-1]$ est constitué des éléments initialement aux positions $[0, \ldots, i-1]$ et est trié par ordre croissant

Preuve d'exactitude du tri par insertion

Invariant du tri par insertion

Le tableau $A[0, \ldots, i-1]$ est constitué des éléments initialement aux positions $[0, \ldots, i-1]$ et est trié par ordre croissant

Preuve d'exactitude

1. **Initialisation**

   L'invariant est vrai puisque lorsque $i == 1$ le tableau $A[0, \ldots, i-1]$ est réduit à $A[0]$ qui est trié

2. **Itération**

   L'invariant reste vrai puisque la boucle **while** interne déplace vers la droite les éléments qui sont plus grands que $A[i]$ et insère la clé, initialement en $A[i]$, à la position correcte

3. **Terminaison**

   La boucle while externe s'arrête lorsque $i == n$
   A ce stade, le tableau $A[0, \ldots, i-1] = A[0, \ldots, n-1]$ est trié
   L'ensemble du tableau est donc trié et l'algorithme est correct

Exactitude de la factorielle par induction

Algorithm: fact
Input: Un entier n
Output: $n!$
if $n == 0$ **then**
  return 1
return $n \times$ `fact(`n-1`)`

Preuve de l'exactitude par induction

1. Cas de base : `fact(`0`)` $= 1 = 0!$
2. Induction :
   - Si `fact(`n-1`)` est correct
   - alors `fact(`n`)` est correct

   `fact(`n`)` $= n \times$ `fact(`n-1`)` $\overset{\text{Hyp.}}{=} n \times (n-1)! = n!$
3. Conclusion : `fact(`n`)` calcule correctement $n!$ pour tout $n \geq 0$

Exactitude de la factorielle par invariant

Algorithm: factoriellelter
Input: Un entier naturel n
Output: $n!$
$fact \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $fact \leftarrow fact \times i$
return $fact$

Preuve de l'exactitude par invariant

Invariant : $fact = (i-1)!$

1. Initialisation :
   Lorsque $i == 2$, $fact = 1 = (2-1)! = 1!$
2. Itération :
   - A chaque exécution

   $fact \leftarrow fact * i = (i-1)! * i = i!$
   $i \leftarrow i+1 \implies fact = (i-1)!$
3. Terminaison :
   - La boucle s'arrête lorsque $i == n+1$
   - A ce stade :
     $fact = (i-1)! = (n+1-1)! = n!$
   - L'algorithme est correct

Résumé

- Un invariant est une propriété qui ne varie pas
- Les invariantes de boucle sont utiles pour démontrer l'exactitude d'algorithmes itératifs
- Les preuves par induction sont utiles pour démontrer l'exactitude d'algorithmes récursifs
- Une preuve par induction généralise pour toute taille n du problème alors que la preuve par invariant considère explicitement la terminaison

Exercices sur les preuves d'algorithmes itératifs et récursifs :

https://inginious.org/course/prepa-master/Algo_23_puissance

https://inginious.org/course/prepa-master/Algo_24_produit

https://inginious.org/course/prepa-master/Algo_25_cumsum_invariant

https://inginious.org/course/prepa-master/Algo_26_exp

https://inginious.org/course/prepa-master/Algo_27_horner