Alain Troesch
Cours d'informatique, MPSI 4
Lycée Louis-Le-Grand (Paris)
Année scolaire 2015/2016

Informatique – Chapitre 4

Algorithmique : terminaison et correction

But

- Rappeler les structures élémentaires constituant un algorithme

But

- Rappeler les structures élémentaires constituant un algorithme
- Analyser la terminaison de l'algorithme

But

- Rappeler les structures élémentaires constituant un algorithme
- Analyser la terminaison de l'algorithme
- Analyse la correction de l'algorithme.

I. Algorithmes

I-1. Définition

Définition 1.1 (Algorithme)

C'est une succession d'instructions élémentaires :

I. Algorithmes

I-1. Définition

Définition 1.1 (Algorithme)

C'est une succession d'instructions élémentaires :

- non ambiguës,

I. Algorithmes

I-1. Définition

Définition 1.1 (Algorithme)

C'est une succession d'instructions élémentaires :

- non ambiguës,
- déterminées de façon unique par les données initiales,

I. Algorithmes

I-1. Définition

Définition 1.1 (Algorithme)

C'est une succession d'instructions élémentaires :

- non ambiguës,
- déterminées de façon unique par les données initiales,
- fournissant la réponse à un problème posé.

I. Algorithmes

I-1. Définition

Définition 1.1 (Algorithme)

C'est une succession d'instructions élémentaires :

- non ambiguës,
- déterminées de façon unique par les données initiales,
- fournissant la réponse à un problème posé.

Etymologie : déformation du nom Al Khwarizmi

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées
- Division euclidenne par différences successives

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées
- Division euclidenne par différences successives
- l'algorithme d'Euclide

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées
- Division euclidenne par différences successives
- l'algorithme d'Euclide
- l'algorithme de résolution des équations de degré 2

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées
- Division euclidenne par différences successives
- l'algorithme d'Euclide
- l'algorithme de résolution des équations de degré 2
- l'algorithme du pivot de Gauss

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées
- Division euclidenne par différences successives
- l'algorithme d'Euclide
- l'algorithme de résolution des équations de degré 2
- l'algorithme du pivot de Gauss
- l'algorithme de Hörner

Les algorithmes existent bien avant l'informatique, et sont à lier à la notion de méthode de calcul ou de résolution :

- Les algorithmes de calcul des opérations élémentaires posées
- Division euclidenne par différences successives
- l'algorithme d'Euclide
- l'algorithme de résolution des équations de degré 2
- l'algorithme du pivot de Gauss
- l'algorithme de Hörner
- etc.

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

Nous décrirons systématiquement un algorithme en :

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales
3. définissant la sortie

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales
3. définissant la sortie
4. donnant le bloc d'instructions définissant l'algorithme.

I-2. Le langage

On utilise un pseudo-code, afin de se dégager des spécificités d'un langage existant.

Nous décrirons systématiquement un algorithme en :

1. lui donnant un nom ;
2. définissant les données initiales
3. définissant la sortie
4. donnant le bloc d'instructions définissant l'algorithme.

La structure générale est donc la suivante :

Algorithme 1 : Nom ou description de l'algorithme

Entrée : a,b,... : type
Sortie : c,d,... : type

instructions ;
renvoyer (c, d,...)

I-3. Les structures élémentaires

1. **La séquence :**

I-3. Les structures élémentaires

1. **La séquence :**
   - Regroupement d'instructions élémentaires

I-3. Les structures élémentaires

1. **La séquence :**
    - Regroupement d'instructions élémentaires
    - Notion de bloc en informatique.

I-3. Les structures élémentaires

1. **La séquence :**
    - Regroupement d'instructions élémentaires
    - Notion de bloc en informatique.
    - délimitation par balises de début et fin ou par indentation.

I-3. Les structures élémentaires

1. **La séquence :**
   - Regroupement d'instructions élémentaires
   - Notion de bloc en informatique.
   - délimitation par balises de début et fin ou par indentation.
   - But : pouvoir considérer plusieurs instructions comme une seule.

I-3. Les structures élémentaires

1. **La séquence :**

   - Regroupement d'instructions élémentaires
   - Notion de bloc en informatique.
   - délimitation par balises de début et fin ou par indentation.
   - But : pouvoir considérer plusieurs instructions comme une seule.

Algorithme 2 : Bloc

```
début bloc
  | instructions
fin bloc
```

I-3. Les structures élémentaires

1. **La séquence :**
   - Regroupement d'instructions élémentaires
   - Notion de bloc en informatique.
   - délimitation par balises de début et fin ou par indentation.
   - But : pouvoir considérer plusieurs instructions comme une seule.

Algorithme 2 : Bloc

```
début bloc
  | instructions
fin bloc
```

La plupart du temps, un bloc permet de délimiter la portée d'une structure composée.

2. La structure conditionnelle simple

2. **La structure conditionnelle simple**

- Pour les disjonctions de cas

2. La structure conditionnelle simple

- Pour les disjonctions de cas
- Branchement à 2 issues :
  `si` *condition* `alors` *instructions* `sinon` *instructions*

2. La structure conditionnelle simple

- Pour les disjonctions de cas
- Branchement à 2 issues :
  `si` *condition* `alors` *instructions* `sinon` *instructions*

Algorithme 3 : Str. cond. simple sans clause alternative

Entrée : classe : entier
Sortie : $\varnothing$

si *classe* $= 4$ **alors**
| `Afficher(`*'Bestial !'*`)`
fin si

2. La structure conditionnelle simple

- Pour les disjonctions de cas
- Branchement à 2 issues :
  `si` *condition* `alors` *instructions* `sinon` *instructions*

Algorithme 4 : Str. cond. simple avec clause alternative

Entrée : classe : entier
Sortie : $\varnothing$

si $classe = 4$ **alors**
| `Afficher(`*'Bestial!'*`)`
sinon
| `Afficher(`*'Khrass'*`)`
fin si

2. La structure conditionnelle simple

- Pour les disjonctions de cas
- Branchement à 2 issues :
  `si` *condition* `alors` *instructions* `sinon` *instructions*
- Certains langages autorisent le branchement multiple

2. La structure conditionnelle simple

- Pour les disjonctions de cas
- Branchement à 2 issues :
  `si` *condition* `alors` *instructions* `sinon` *instructions*
- Certains langages autorisent le branchement multiple

Algorithme 5 : Structure conditionnelle multiple

Entrée : classe : entier
Sortie : $\varnothing$

si *classe* $= 4$ **alors**
| `Afficher`(*'Bestial !'*)
sinon si *classe* $= 3$ **alors**
| `Afficher`(*'Skiii !'*)
sinon
| `Afficher`(*'Khrass'*)
fin si

2. La structure conditionnelle simple

- Pour les disjonctions de cas
- Branchement à 2 issues :
  `si` *condition* `alors` *instructions* `sinon` *instructions*
- Certains langages autorisent le branchement multiple

Algorithme 6 : Version équivalente

Entrée : classe : entier
Sortie : $\varnothing$
si *classe* $= 4$ **alors**
    `Afficher`(*'Bestial!'*)
sinon
    si *classe* $= 3$ **alors**
        `Afficher`(*'Skiii!'*)
    sinon
        `Afficher`(*'Khrass'*)
    fin si
fin si

3. Les boucles conditionnelles `while`

- Une boucle est une succession d'instructions, répétée un certain nombre de fois.
- Boucle conditionnelle while : la condition de continuation dépend du résultat d'un test.

3. Les boucles conditionnelles `while`

- Une boucle est une succession d'instructions, répétée un certain nombre de fois.
- Boucle conditionnelle while : la condition de continuation dépend du résultat d'un test.

3. **Les boucles conditionnelles `while`**

- Une boucle est une succession d'instructions, répétée un certain nombre de fois.
- Boucle conditionnelle while : la condition de continuation dépend du résultat d'un test.
- En général, on ne sait pas initialement combien de passages dans la boucle.

3. Les boucles conditionnelles `while`

- Une boucle est une succession d'instructions, répétée un certain nombre de fois.
- Boucle conditionnelle while : la condition de continuation dépend du résultat d'un test.
- En général, on ne sait pas initialement combien de passages dans la boucle.

Algorithme 7 : Que fait cet algorithme ?

Entrée : ε (marge d'erreur) : réel
Sortie : $\varnothing$

$u \leftarrow 1$;
tant que $u > \varepsilon$ **faire**
  $u \leftarrow \sin(u)$
fin tant que

3. Les boucles conditionnelles `while`

- Si on est intéressé par le nombre d'itération : créer un compteur.

3. Les boucles conditionnelles `while`

- Si on est intéressé par le nombre d'itération : créer un compteur.

Algorithme 8 : Calcul de i tel que $u_i \leqslant \varepsilon$

Entrée : ε (marge d'erreur) : réel
Sortie : i : entier

$u \leftarrow 1$;
$i \leftarrow 0$;
tant que $u > \varepsilon$ **faire**
  $u \leftarrow \sin(u)$;
  $i \leftarrow i + 1$
fin tant que
renvoyer i

4. Boucles conditionnelles `repeat`

- Même principe mais avec une condition d'arrêt

4. Boucles conditionnelles `repeat`

- Même principe mais avec une condition d'arrêt

Algorithme 9 : Vitesse de convergence de u_n

Entrée : ε (marge d'erreur) : réel
Sortie : i (rang tel que $u_i \leqslant \varepsilon$) : entier

$u \leftarrow 0$;
$i \leftarrow 0$;
répéter
    $u \leftarrow \sin(u)$;
    $i \leftarrow i + 1$
jusqu'à ce que $u \leqslant \varepsilon$;
renvoyer i;

4. Boucles conditionnelles `repeat`

- Même principe mais avec une condition d'arrêt
- Différence avec `while` : on passe au moins une fois dans la boucle

Algorithme 9 : Vitesse de convergence de u_n

Entrée : ε (marge d'erreur) : réel
Sortie : i (rang tel que $u_i \leqslant \varepsilon$) : entier

$u \leftarrow 0$;
$i \leftarrow 0$;
répéter
  $u \leftarrow \sin(u)$;
  $i \leftarrow i + 1$
jusqu'à ce que $u \leqslant \varepsilon$;
renvoyer i;

4. Boucles conditionnelles `repeat`

- Une boucle `repeat` est équivalente à :

4. Boucles conditionnelles `repeat`

- Une boucle `repeat` est équivalente à :

Algorithme 10 : équivalent à `repeat... until...` (1)

instructions ;
tant que $\neg$ *condition* **faire**
  instructions
fin tant que

4. Boucles conditionnelles `repeat`

- Une boucle `repeat` est équivalente à :

Algorithme 11 : équivalent à `repeat... until...` (2)

$b \leftarrow$ True ;
tant que *($\neg$ condition)* $\vee$ b **faire**
  instructions ;
  $b \leftarrow$ False
fin tant que

---

4. Boucles conditionnelles `repeat`

- Une boucle `repeat` est équivalente à :
- Une boucle `while` est équivalente à :

4. Boucles conditionnelles `repeat`

- Une boucle `repeat` est équivalente à :
- Une boucle `while` est équivalente à :

Algorithme 12 : Structure équivalente à `while... do...`

si *condition* **alors**
  répéter
    instructions
  jusqu'à ce que $\neg$ *condition*;
fin si

4. Boucles conditionnelles `repeat`

- Une boucle `repeat` est équivalente à :
- Une boucle `while` est équivalente à :
- Nous nous autoriserons les 2 structures, même si en Python il n'y en a qu'une.

5. Boucles inconditionnelles `for`

- Le nombre d'itérations est connu initialement.

5. Boucles inconditionnelles `for`

- Le nombre d'itérations est connu initialement.
- Pas de test d'arrêt, mais on compte les passages.

5. Boucles inconditionnelles `for`

- Le nombre d'itérations est connu initialement.
- Pas de test d'arrêt, mais on compte les passages.

Algorithme 13 : Cri de guerre

Entrée : $\varnothing$
Sortie : cri : chaîne de caractères

bestial $\leftarrow$ 'besti' ;
pour $i \leftarrow 1$ **à** 42 **faire**
| bestial $\leftarrow$ bestial + 'â'
fin pour
bestial $\leftarrow$ bestial + 'l' ;
renvoyer *cri*

I-4. Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme.

I-4. Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme.

- Éviter d'avoir à réécrire plusieurs fois la même séquence

I-4. Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme.

- Éviter d'avoir à réécrire plusieurs fois la même séquence
- Rendre une séquence plus modulable (par les paramètres)

I-4. Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme.

- Éviter d'avoir à réécrire plusieurs fois la même séquence
- Rendre une séquence plus modulable (par les paramètres)
- Dégager le programme de toute la partie technique

I-4. Procédures, fonctions et récursivité

Un algorithme peut faire appel à des sous-algorithmes, ou procédures, qui sont des morceaux isolés de programme.

- Éviter d'avoir à réécrire plusieurs fois la même séquence
- Rendre une séquence plus modulable (par les paramètres)
- Dégager le programme de toute la partie technique
- Définir des algorithmes récursifs.

Procédure 14 : affichepolynome(T)

```
ch ← '' ;
pour i ← Taille (T)-1 descendant à 0 faire
    si T[i] ≠ 0 alors
        si (T[i] > 0) alors
            si ch ≠ '' alors
                ch ← ch + ' + '
            fin si
        sinon
            ch ← ch + ' - '
        fin si
        si (|T[i]| ≠ 1) ∨ (i = 0) alors
            ch ← ch + str(|T[i]|);
            si i ≠ 0 alors
                ch ← ch + ' * '
            fin si
        fin si
        si i ≠ 0 alors
            ch ← ch + 'X'+ '^' + str(i)
        fin si
    fin si
fin pour
Afficher (ch)
```

Une fonction est similaire à une procédure, mais renvoie en plus une valeur de sortie. Par exemple :

Une fonction est similaire à une procédure, mais renvoie en plus une valeur de sortie. Par exemple :

Fonction 15 : comptea(ch)

Entrée : ch : chaîne de caractères
Sortie : n : entier

$n \leftarrow 0$;
pour $i \leftarrow 0$ **à** `Taille`(ch) **faire**
    si $\mathrm{ch}[i] =$ *'a'* **alors**
        $n \leftarrow n + 1$
    fin si
fin pour
renvoyer n

- Une fonction récursive est une fonction qui s'appelle elle-même.

- Une fonction récursive est une fonction qui s'appelle elle-même.

Fonction 16 : un(n)

Entrée : n : entier positif
Sortie : un(n) : réel

si $n = 0$ **alors**
  renvoyer 1
sinon
  renvoyer sin$(\mathrm{un}(n-1))$
fin si

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Il faut faire attention à ce qu'elle définisse les conditions initiales.

Fonction 16 : un(n)

Entrée : n : entier positif
Sortie : un(n) : réel

si $n = 0$ **alors**
| **renvoyer** 1
sinon
| **renvoyer** sin($\mathrm{un}(n-1)$)
fin si

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Il faut faire attention à ce qu'elle définisse les conditions initiales.
- Attention aux risques d'explosion en complexité !

- Une fonction récursive est une fonction qui s'appelle elle-même.
- Il faut faire attention à ce qu'elle définisse les conditions initiales.
- Attention aux risques d'explosion en complexité !

Fonction 17 : fibo(n)

Entrée : n : entier positif
Sortie : fibo(n) : réel

si $n = 0$ **alors**
  | **renvoyer** 0
sinon si $n = 1$ **alors**
  | **renvoyer** 1
sinon
  | **renvoyer** $\mathrm{fibo}(n-1) + \mathrm{fibo}(n-2)$
fin si

II. Validité d'un algorithme

Deux questions peuvent se poser :

II. Validité d'un algorithme

Deux questions peuvent se poser :

- L'algorithme s'arrête-t-il ? (problème de terminaison)

II. Validité d'un algorithme

Deux questions peuvent se poser :

- L'algorithme s'arrête-t-il ? (problème de terminaison)
- L'algorithme renvoie-t-il le résultat attendu ? (problème de correction).

II-1. Terminaison d'un algorithme

Dans un algorithme non récursif, le seul cas possible de non terminaison provient de `while` ou `repeat`.

II-1. Terminaison d'un algorithme

Dans un algorithme non récursif, le seul cas possible de non terminaison provient de `while` ou `repeat`.

L'archétype de la preuve de terminaison est :

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
 $(a, b) \leftarrow (b, a \mod b)$
sinon
 renvoyer a
fin si

II-1. Terminaison d'un algorithme

Dans un algorithme non récursif, le seul cas possible de non terminaison provient de `while` ou `repeat`.

L'archétype de la preuve de terminaison est :

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
  $(a, b) \leftarrow (b, a \mod b)$
sinon
  renvoyer a
fin si

Principe : descente infinie

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \dots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \dots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;
- v (en entrée de boucle) soit toujours positive ;

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \ldots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;
- v (en entrée de boucle) soit toujours positive ;
- v décroît strictement.

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \ldots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;
- v (en entrée de boucle) soit toujours positive ;
- v décroît strictement.

Théorème 2.2 (Terminaison d'une boucle, d'un algorithme)

1. Une boucle possédant un variant de boucle se finit

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \ldots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;
- v (en entrée de boucle) soit toujours positive ;
- v décroît strictement.

Théorème 2.2 (Terminaison d'une boucle, d'un algorithme)

1. Une boucle possédant un variant de boucle se finit
2. Un algorithme dont toutes les boucles possèdent un variant se finit.

Définition 2.1 (Variant de boucle)

variant de boucle : quantité v dépendant de $(x_1, \ldots, x_k)$ et n (nombre d'itérations), telle que :

- v ne prenne que des valeurs entières ;
- v (en entrée de boucle) soit toujours positive ;
- v décroît strictement.

Théorème 2.2 (Terminaison d'une boucle, d'un algorithme)

1. Une boucle possédant un variant de boucle se finit
2. Un algorithme dont toutes les boucles possèdent un variant se finit.

Avertissement 2.3

Doit être valable pour toute entrée possible.

Exemple 2.4

Dans le cas de l'algorithme d'Euclide : b est un variant de boucle (à partir du rang 1)

Exemple 2.4

Dans le cas de l'algorithme d'Euclide : b est un variant de boucle (à partir du rang 1)

Remarque 2.5

- La terminaison assure un arrêt théorique de l'algorithme.

Exemple 2.4

Dans le cas de l'algorithme d'Euclide : b est un variant de boucle (à partir du rang 1)

Remarque 2.5

- La terminaison assure un arrêt théorique de l'algorithme.
- Cela ne tient pas compte des problèmes de complexité : un arrêt théorique en quelques milliards d'années est d'un intérêt limité.

Exemple 2.4

Dans le cas de l'algorithme d'Euclide : b est un variant de boucle (à partir du rang 1)

Remarque 2.5

- La terminaison assure un arrêt théorique de l'algorithme.
- Cela ne tient pas compte des problèmes de complexité : un arrêt théorique en quelques milliards d'années est d'un intérêt limité.

Remarque 2.6

Dans le cas d'une boucle `Pour` $i \leftarrow a$ à b, un variant simple est $b - i$. Tout boucle `for` se finit.

Algorithme 19 : Mystère

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant b$ **faire**
    $r \leftarrow r - b$;
    $q \leftarrow q + 1$;
  fin tant que
  renvoyer q,r
fin si

Algorithme 19 : Mystère

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant b$ **faire**
    $r \leftarrow r - b$;
    $q \leftarrow q + 1$;
  fin tant que
  renvoyer q,r
fin si

Variant de boucle : $v = r - b$?

Algorithme 19 : Mystère

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
    Erreur
sinon
    tant que $r \geqslant b$ **faire**
        $r \leftarrow r - b$;
        $q \leftarrow q + 1$;
    fin tant que
    renvoyer q,r
fin si

Variant de boucle : $v = r - b$?
Problème pour $b < 0$.

Algorithme 20 : Division euclidienne ?

Entrée : a,b : entiers
Sortie : q, r : entiers

```
r ← a ;
q ← 0 ;
si b = 0 alors
    Erreur
sinon
    tant que r ⩾ |b| faire
        r ← r − |b| ;
        q ← q + 1 ;
    fin tant que
    si b < 0 alors
        q ← −q
    fin si
    renvoyer q,r
fin si
```

Algorithme 20 : Division euclidienne ?

```
Entrée : a,b : entiers
Sortie : q, r : entiers

r ← a ;
q ← 0 ;
si b = 0 alors
    Erreur
sinon
    tant que r ⩾ |b| faire
        r ← r − |b| ;
        q ← q + 1 ;
    fin tant que
    si b < 0 alors
        q ← −q
    fin si
    renvoyer q,r
fin si
```

$v = r - |b|$ est un variant de boucle, ce qui prouve la terminaison

II-2. Correction d'un algorithme

- La terminaison n'assure pas la validité du résultat !

II-2. Correction d'un algorithme

- La terminaison n'assure pas la validité du résultat !
- Validité de l'algorithme d'Euclide : $a \wedge b = b \wedge r$.

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
  $(a, b) \leftarrow (b, a \mod b)$
sinon
  renvoyer a
fin si

II-2. Correction d'un algorithme

- La terminaison n'assure pas la validité du résultat !
- Validité de l'algorithme d'Euclide : $a \wedge b = b \wedge r$.
- Valeurs successives de $a \wedge b$ en entrée d'itération = cste,

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
  $(a, b) \leftarrow (b, a \mod b)$
sinon
  renvoyer a
fin si

II-2. Correction d'un algorithme

- La terminaison n'assure pas la validité du résultat !
- Validité de l'algorithme d'Euclide : $a \wedge b = b \wedge r$.
- Valeurs successives de $a \wedge b$ en entrée d'itération = cste,
- En sortie de boucle $a \wedge b$ est égal à a

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
  $(a, b) \leftarrow (b, a \mod b)$
sinon
  renvoyer a
fin si

II-2. Correction d'un algorithme

- La terminaison n'assure pas la validité du résultat !
- Validité de l'algorithme d'Euclide : $a \wedge b = b \wedge r$.
- Valeurs successives de $a \wedge b$ en entrée d'itération $=$ cste,
- En sortie de boucle $a \wedge b$ est égal à a
- Ainsi, la valeur finale de a est bien $a \wedge b$ initial.

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
  $(a, b) \leftarrow (b, a \mod b)$
sinon
  renvoyer a
fin si

II-2. Correction d'un algorithme

- La terminaison n'assure pas la validité du résultat !
- Validité de l'algorithme d'Euclide : $a \wedge b = b \wedge r$.
- Valeurs successives de $a \wedge b$ en entrée d'itération $=$ cste,
- En sortie de boucle $a \wedge b$ est égal à a
- Ainsi, la valeur finale de a est bien $a \wedge b$ initial.
- On dit que $a \wedge b$ est un invariant de boucle.

Algorithme 18 : Euclide

Entrée : a, b : entiers
Sortie : d : entier

si $b \neq 0$ **alors**
  $(a, b) \leftarrow (b, a \mod b)$
sinon
  renvoyer a
fin si

Définition 2.7 (Invariant de boucle)

Invariant de boucle : quantité w dépendant des $x_1, \ldots, x_k$ et de n, nombre d'itérations, telle que :

Définition 2.7 (Invariant de boucle)

Invariant de boucle : quantité w dépendant des $x_1, \ldots, x_k$ et de n, nombre d'itérations, telle que :

- w (en entrée d'itération) $=$ constante

Définition 2.7 (Invariant de boucle)

Invariant de boucle : quantité w dépendant des $x_1, \ldots, x_k$ et de n, nombre d'itérations, telle que :

- w (en entrée d'itération) $=$ constante
- l'égalité $w_0 = w_n$ permet de prouver que l'algorithme retourne le bon résultat.

Définition 2.7 (Invariant de boucle)

Invariant de boucle : quantité w dépendant des $x_1, \dots, x_k$ et de n, nombre d'itérations, telle que :

- w (en entrée d'itération) $=$ constante
- l'égalité $w_0 = w_n$ permet de prouver que l'algorithme retourne le bon résultat.

Exemple 2.8

Invariant de boucle pour l'algorithme d'Euclide : $w = a \wedge b$

Algorithme 20 : Division euclidienne ?

```
Entrée : a,b : entiers
Sortie : q, r : entiers

r ← a ;
q ← 0 ;
si b = 0 alors
    Erreur
sinon
    tant que r ⩾ |b| faire
        r ← r − |b| ;
        q ← q + 1 ;
    fin tant que
    si b < 0 alors
        q ← −q
    fin si
    renvoyer q,r
fin si
```

Algorithme 20 : Division euclidienne ?

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant |b|$ **faire**
    $r \leftarrow r - |b|$;
    $q \leftarrow q + 1$;
  fin tant que
  si $b < 0$ **alors**
    $q \leftarrow -q$
  fin si
  renvoyer q,r
fin si

Invariant de boucle $|b|q + r$?

Algorithme 20 : Division euclidienne ?

Entrée : a,b : entiers
Sortie : q, r : entiers

$r \leftarrow a$;
$q \leftarrow 0$;
si $b = 0$ **alors**
  Erreur
sinon
  tant que $r \geqslant |b|$ **faire**
    $r \leftarrow r - |b|$;
    $q \leftarrow q + 1$;
  fin tant que
  si $b < 0$ **alors**
    $q \leftarrow -q$
  fin si
  renvoyer q,r
fin si

Invariant de boucle $|b|q + r$? constant mais pb si $a \leqslant 0$!

Algorithme 21 : Division euclidienne, version corrigée

```
Évacuer le cas b = 0 ;
si a ⩾ 0 alors
    tant que r ⩾ |b| faire
        r ← r − |b| ;
        q ← q + 1 ;
    fin tant que
    si b < 0 alors
        q ← −q
    fin si
    renvoyer q,r
sinon
    tant que r < 0 faire
        r ← r + |b| ;
        q ← q − 1 ;
    fin tant que
    si b < 0 alors
        q ← −q
    fin si
    renvoyer q,r
fin si
```

Justification de la terminaison et de la correction de cet algorithme

1. Cas $a \geqslant 0$:

Justification de la terminaison et de la correction de cet algorithme

1. Cas $a \geqslant 0$:
   - Variant : $r - |b|$

Justification de la terminaison et de la correction de cet algorithme

1. Cas $a \geqslant 0$:
   - Variant : $r - |b|$
   - Invariant : $|b|q + r$.

Justification de la terminaison et de la correction de cet algorithme

1. Cas $a \geqslant 0$:
   - Variant : $r - |b|$
   - Invariant : $|b|q + r$.
2. Cas $a < 0$:

Justification de la terminaison et de la correction de cet algorithme

1. Cas $a \geqslant 0$:
   - Variant : $r - |b|$
   - Invariant : $|b|q + r$.
2. Cas $a < 0$:
   - Variant : $-r$

Justification de la terminaison et de la correction de cet algorithme

1. Cas $a \geqslant 0$:
   - Variant : $r - |b|$
   - Invariant : $|b|q + r$.
2. Cas $a < 0$:
   - Variant : $-r$
   - Invariant : $|b|q + r$.

Remarque 2.9

Invariant de boucle booléen = véracité d'une propriété $\mathcal{P}(n)$.

Remarque 2.9

Invariant de boucle booléen = véracité d'une propriété $\mathcal{P}(n)$.

Algorithme 22 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
    si $T[i] < T[0]$ **alors**
        $T[0], T[i] \leftarrow T[i], T[0]$
    fin si
fin pour

Remarque 2.9

Invariant de boucle booléen = véracité d'une propriété $\mathcal{P}(n)$.

Algorithme 22 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
  si $T[i] < T[0]$ **alors**
    $T[0], T[i] \leftarrow T[i], T[0]$
  fin si
fin pour

$\mathcal{P}(n)$: à l'entrée au rang $n \geqslant 1$, $T[0] = min(T[i], i \in [\![0, n-1]\!])$.

Remarque 2.9

Invariant de boucle booléen = véracité d'une propriété $\mathcal{P}(n)$.

Algorithme 22 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
  si $T[i] < T[0]$ **alors**
    $T[0], T[i] \leftarrow T[i], T[0]$
  fin si
fin pour

$\mathcal{P}(n)$: à l'entrée au rang $n \geqslant 1$, $T[0] = min(T[i], i \in [\![0, n-1]\!])$.

- $\mathcal{P}(1)$ vrai

Remarque 2.9

Invariant de boucle booléen = véracité d'une propriété $\mathcal{P}(n)$.

Algorithme 22 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
  si $T[i] < T[0]$ **alors**
    $T[0], T[i] \leftarrow T[i], T[0]$
  fin si
fin pour

$\mathcal{P}(n)$: à l'entrée au rang $n \geqslant 1$, $T[0] = min(T[i], i \in [\![0, n-1]\!])$.

- $\mathcal{P}(1)$ vrai
- $\mathcal{P}(n)$ vrai $\Longrightarrow \mathcal{P}(n+1)$ vrai

Remarque 2.9

Invariant de boucle booléen = véracité d'une propriété $\mathcal{P}(n)$.

Algorithme 22 : Selection du minimum

Entrée : T : tableau
Sortie : T :tableau

pour $i \leftarrow 1$ **à** `Taille` *(T) - 1* **faire**
  si $T[i] < T[0]$ **alors**
    $T[0], T[i] \leftarrow T[i], T[0]$
  fin si
fin pour

$\mathcal{P}(n)$: à l'entrée au rang $n \geqslant 1$, $T[0] = min(T[i], i \in [\![0, n-1]\!])$.

- $\mathcal{P}(1)$ vrai
- $\mathcal{P}(n)$ vrai $\Longrightarrow \mathcal{P}(n+1)$ vrai
- la véracité de $\mathcal{P}(\texttt{Taille(T)})$ implique le résultat attendu.

Dans cette situation, montrer la correction d'un algorithme se fait en 3 étapes :

Dans cette situation, montrer la correction d'un algorithme se fait en 3 étapes :

- initialisation : montrer que l'invariant est vrai avant la première itération.

Dans cette situation, montrer la correction d'un algorithme se fait en 3 étapes :

- initialisation : montrer que l'invariant est vrai avant la première itération.
- conservation : montrer que si l'invariant est vrai avant une itération, il reste vrai avant l'itération suivante.

Dans cette situation, montrer la correction d'un algorithme se fait en 3 étapes :

- initialisation : montrer que l'invariant est vrai avant la première itération.
- conservation : montrer que si l'invariant est vrai avant une itération, il reste vrai avant l'itération suivante.
- terminaison : déduire de l'invariant final la propriété voulue.

III. Exercices

Exercice 1 (recherche linéaire)

Écrire un algorithme en pseudo-code prenant en entrée une liste L, et une valeur v, et donnant en sortie un indice i tel que $L[i] = v$, s'il en existe, et la valeur spéciale `NIL` sinon. Étudier la correction et la terminaison de cet algorithme.

Exercice 2 (Tri à bulle)

Étudier la correction et la terminaison de l'algorithme ci-dessous. Proposer une amélioration repérant lors du passage dans la boucle interne, le plus grand indice pour lequel une inversion a été faite. Jusqu'où suffit-il d'aller pour la boucle suivante ? Étudier la correction et la terminaison du nouvel algorithme obtenu.

Algorithme 23 : Tri-bulle

Entrée : T :tableau de réels, n :taille du tableau
Sortie : T : tableau trié
pour $i \leftarrow 1$ **à** $n-1$ **faire**
    pour $j \leftarrow 1$ **à** $n-i$ **faire**
        si $T[j] < T[j-1]$ **alors**
            $T[j-1], T[j] \leftarrow T[j], T[j-1]$
        fin si
    fin pour
fin pour

Exercice 3 (Tri par sélection)

Donner le pseudo-code de l'algorithme de tri par sélection consistant à rechercher le minimum et à le mettre à sa place, puis à rechercher le deuxième élément et à le mettre à sa place etc. Étudier la terminaison et la correction de cet algorithme.

Exercice 4 (Exponentiation rapide)

Comprendre ce que fait l'algorithme suivant. Étudier sa terminaison et sa correction. Écrire un algorithme récursif d'exponentiation rapide.

Algorithme 24 : Exponentiation rapide

Entrée : x : réel, n :entier
$u \leftarrow 1$;
$v \leftarrow 1$;
tant que $n > 1$ **faire**
  $(n, b) \leftarrow \texttt{divmod}(n, 2)$;
  si $b = 1$ **alors**
    $v \leftarrow v \times u$
  fin si
  $u \leftarrow u^2$
fin tant que
renvoyer $u \times v$

Exercice 5 (Décomposition)

Que fait cet algorithme ? Justifier sa terminaison et sa correction.

Algorithme 25 : Décomposition

Entrée : n : entier positif
Sortie : L : liste de couples d'entier ; à quoi correspond-elle ?

```
p ← 2 ; L ← [] ;
tant que p ⩽ √n faire
    i ← 0 ;
    tant que n mod p = 0 faire
        i ← i + 1 ; n ← n/p
    fin tant que
    si i > 0 alors
        Ajouter (p, i) à la liste L
    fin si
    si p = 2 alors
        p ← p + 1
    sinon
        p ← p + 2
    fin si
fin tant que
si n > 1 alors
    Ajouter (n, 1) à la liste L
fin si
```