Structures répétitives [lp]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 17 mai 2018

Table des matières

Python - Structures répétitives (Cours)

Mots-Clés Structures répétitives, Ruptures de blocs ■
Requis Structures de base, Structures conditionnelles ■
Difficulté ● ● ○

Introduction
Ce module introduit la notion de travail répétitif puis décrit les structures **répétitives** `TantQue`, `Répéter`, `Itérer` et `Pour`. La *Synthèse* précise les pièges à éviter et donne les méthodes d'écriture utilisant uniquement la structure `TantQue`. Le *Compléments* présente les ruptures de blocs et la section *Approfondir* décrit le « squelette » des suites.

Accrochez-vous...
D'expérience, beaucoup d'entre vous perdent pied ici...
Module difficile à appréhender, accrochez-vous.

Conseil
Faites bien tous les exercices proposés et demandez de l'aide dès que vous vous sentez perdu !

1 La notion de travail répétitif

Si on veut faire effectuer un travail répétitif, il faut indiquer deux choses :

1. Le travail à répéter
2. La manière de continuer la répétition ou de l'arrêter.

Exemple
Pour traiter des dossiers, on dira quelque chose comme :

« tant qu'il reste un dossier à traiter, le traiter »

ou encore :

« traiter un dossier puis passer au suivant
jusqu'à ce qu'il n'en reste plus à traiter »

- Tâche à répéter : « traiter un dossier »
- On indique qu'on doit continuer s'il reste encore un dossier à traiter.

Exemple
Pour calculer la note finale de tous les étudiants, on peut dire :

« Pour tout étudiant, calculer sa note »

- Tâche à répéter : « calculer la note d'un étudiant »
- On indique qu'on doit le faire pour tous les étudiants. On doit donc commencer par le premier, passer à chaque fois au suivant et s'arrêter quand on a fini le dernier.

Exemple
Pour afficher tous les nombres de 1 à 100, on aura :

« Pour tous les nombres de 1 à 100, afficher ce nombre ».

- Tâche à répéter : « afficher un nombre »
- On indique qu'on doit le faire pour tous les nombres de 1 à 100. On doit donc commencer avec 1, passer à chaque fois au nombre suivant et s'arrêter après avoir affiché le nombre 100.

Conclusion
C'est **toujours la même tâche** qui est exécutée **mais avec un effet différent** à chaque fois. Ainsi :

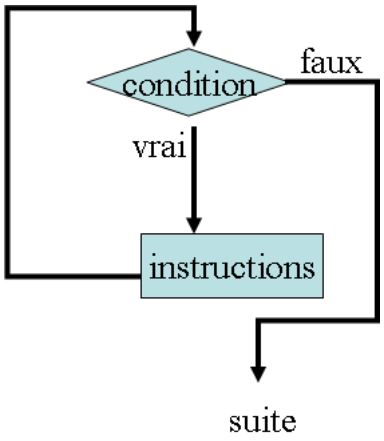
- On traite un dossier, mais à chaque fois un différent.
- On calcule la note d'un étudiant, mais à chaque fois un différent.
- On affiche un nombre, mais à chaque fois un différent.

2 Structures répétitives

2.1 Répétitive TantQue

Répétitive TantQue (répétition a-priori)

Elle traduit : `TantQue` la `condition` est vraie, exécuter les `instructions`.
Si la `condition` est fausse dès le début, la tâche n'est jamais exécutée.



Boucle infinie

La séquence `instructions` **doit** modifier la condition de telle manière qu'elle puisse devenir **fausse**. Dans le cas contraire, la boucle va tourner sans fin (condition indéfiniment vraie) : c'est ce qu'on appelle une **boucle infinie**.

Répétitive TantQue

```
while condition:
  instructions
```

2.2 Exemple : Répétitive Tantque

Programme

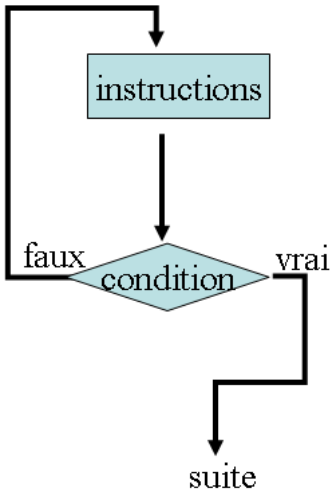
```
sentinelle = ...

def QZTantque1():
    n = int(input("n? "))
    while n != sentinelle:
        traiter_la_valeur_lue_n
        n = int(input("n? "))

QZTantque1()
```

Explication

Le programme exploite une structure `TantQue` afin de traiter un entier `n` tant qu'il est différent de la valeur `sentinelle`.

2.3 Répétitive Répéter

Répétitive Répéter (répétition a-posteriori)
Elle traduit : Exécuter les `instructions` `Jusqu'à` ce que la `condition` est vraie.



Boucle infinie
La séquence `instructions` **doit** modifier la condition de telle manière qu'elle puisse devenir **vraie** pour arrêter l'itération.

Répétitive Répéter

```
while True:
  ...
  if not condition: break
...
```

Python
La répétitive `Répéter` n'existe pas en Python. Elle sera simulée par le code défini ci-avant en utilisant le `TantQue` et l'instruction de rupture `SortirSi`.

2.4 Exemple : Répétitive Répéter

Programme

```
def QZRepeter1():
    while True:
        n = int(input("Entier dans [-3..10]? "))
        if -3 <= n <= 10: break
    print("L'entier saisi est ", n, sep="")

QZRepeter1()
```

Explication

Le programme exploite une structure `Répéter` afin de demander et saisir un entier dans `n` jusqu'à ce qu'il soit dans l'intervalle d'entiers `[a..b]` (ici [-3..10]).

commentée

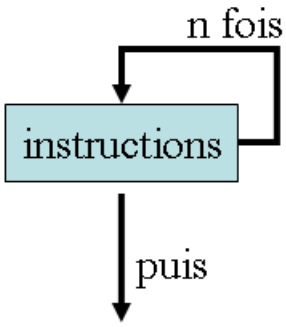
La structure `Répéter` est généralement préférée à la structure `TantQue` lorsque les variables dont dépend la condition reçoivent leur valeur dans la séquence d'instructions, ce qui exige obligatoirement une première itération avant de vérifier la condition. C'est le cas lors d'une saisie validée.
Exemple d'exécution :

```
Entier dans [-3..10]? -5
Entier dans [-3..10]? 15
Entier dans [-3..10]? 2
L'entier saisi est 2
```

2.5 Répétitive Itérer

Répétitive Itérer

Elle traduit : Exécuter `n` fois les `instructions`, avec `n` un entier positif.
Finitude assurée.



Répétitive Itérer

```
for j in range(1,n+1):
   instructions
```

2.6 Exemple : Répétitive Itérer

Programme

```
def QZIterer1():
    for j in range(100):
        print("En classe, je ne bavarde pas avec mon voisin...")

QZIterer1()
```

Explication

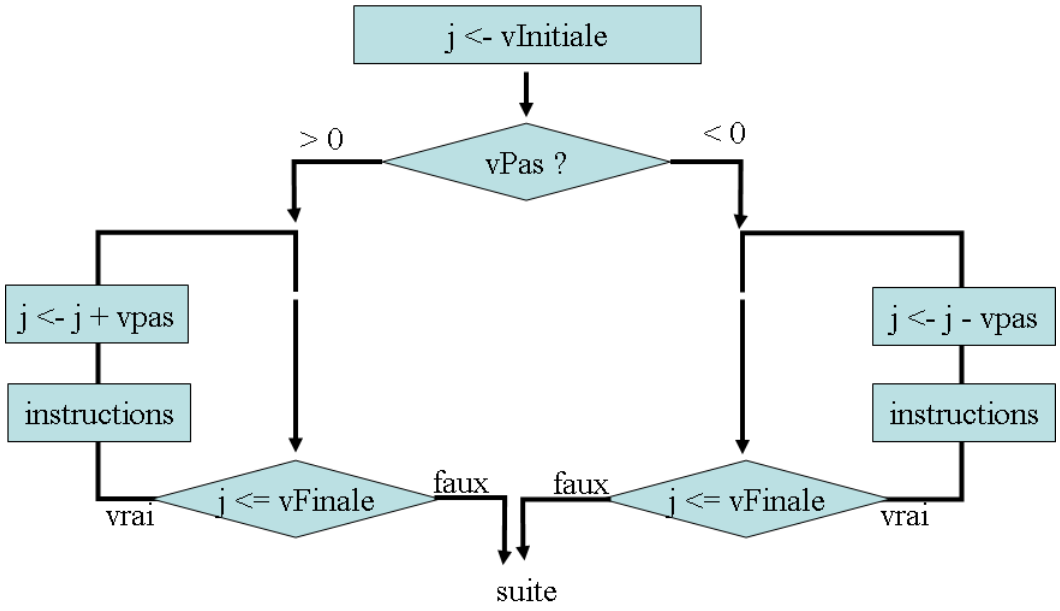
Le programme exploite une structure `Itérer` pour afficher 100 fois le texte :

```
En classe, je ne bavarde pas avec mon voisin...
```

2.7 Répétitive Pour

Répétitive Pour

Elle traduit : Exécuter les `instructions`, `Pour` une variable de boucle `nomVar` (entière ou réelle) dont le contenu varie de la valeur initiale `valDeb` à la valeur finale `valFin` par pas de `valPas` (par défaut de 1). Finitude assurée.



Terminologie

La variable utilisée dans la boucle `Pour` s'appelle la **variable de boucle**, **variable de contrôle**, **indice d'itération** ou **compteur de boucle**. En général, son nom se réduit simplement à une lettre, par exemple `j`.

Répétitive Pour

```
for nomVar in range([valDeb,] valFin+/-1 [, valPas]):
  instructions
```

Attention

Par défaut l'entier `valDeb` vaut 0 et `valPas` vaut 1. L'entier `valFin` (deuxième paramètre) est exclu. Ainsi `range(1,11)` génère les entiers de 1 à 10.

2.8 Exemple : Répétitive Pour

Programme

```
def QZPour1():
    for j in range(1, 4+1):
        print(j, " ", sep="", end="")
    print()
    for j in range(5, 100+1, 10):
        print(j, " ", sep="", end="")
    print()

QZPour1()
```

Explication

Le programme exploite deux structures `Pour` en séquence :

1. Affichez les entiers de 0 à 4.
2. Puis les entiers de 5 à 100 par pas de 10.

La variable `j` est initialisée à 0 avant la première itération, puis successivement augmentée de 1 à la fin de chaque itération (1 est l'incrément par défaut si aucune valeur d'incrémentation n'est fournie).

Lorsque `j` atteint la valeur 5, le flux d'exécution quitte la première structure `Pour` et poursuit l'exécution après le `FinPour`.

2.9 Répétitive imbriquée

Répétitive imbriquée

Boucle (`Pour`, `TantQue`, etc.) contenue dans le corps d'une boucle.

Répétitive Pour-imbriquée

```
for j in range([jdebut,] jfinal+1 [, jpas]):
   for k in range([kdebut,] kfinal+1 [, kpas]):
      instructions
```

Compteurs, ordre d'exécution

Il faut utiliser des **compteurs différents** et faire attention à l'ordre d'exécution des instructions : à chaque passage dans le corps de la boucle extérieure (compteur `j`), la boucle imbriquée (compteur `k`) va être exécutée totalement.

2.10 Exemple : Répétitive Pour-imbriquée

Programme @[pgtriangle1.py]

```
def PGTriangle1():
    n = int(input("Nombre de lignes? "))
    for j in range(1, n+1):
        for k in range(1, j+1):
            print(k % 10, end="")
        print()

PGTriangle1()
```

Explication

La boucle `Pour` imbriquée affiche le triangle de chiffres de hauteur `n` (entier). L'entier `j` est l'indice d'itération sur les lignes et l'entier `k` celui sur les éléments de la `j`-ème ligne.

L'opération `mod` a pour effet de n'afficher que le dernier chiffre de l'entier `k` (au cas où celui-ci serait supérieur à 10).

Exemple d'exécution

```
Nombre de lignes? 5
1
12
123
1234
12345
```

3 Synthèse sur les boucles

3.1 Synthèse sur les boucles

Terminologie

Puisque qu'une exécution de la séquence d'instructions dans une structure répétitive est dite une **itération**, les structures répétitives sont aussi appelées **structures itératives** : ce sont des synonymes.

Boucles inconditionnelles : Itérer, Pour

Conceptuellement, on dit que les boucles `Itérer` et `Pour` sont **inconditionnelles** ou **déterministes** car le nombre d'itérations qu'elles effectuent est prédéterminée et ne dépend pas de la séquence d'instructions dans la boucle.

Boucles conditionnelles : TantQue, Répéter

Les boucles `TantQue` et `Répéter` sont catégorisées comme **conditionnelles** ou **non déterministes** car le nombre d'itérations qu'elles effectuent dépend d'une condition dont la valeur est généralement déterminée par la séquence d'instructions dans la boucle.

Pièges d'une boucle conditionnelle

Vérifiez toujours les points suivants :

- Il est possible d'entrer dans la boucle, ceci afin d'éviter les boucles inutiles.
- Les variables intervenant dans le test de la condition sont correctement initialisées.
- Le corps de la boucle fait évoluer les variables intervenant dans la condition afin d'éviter les boucles infinies.

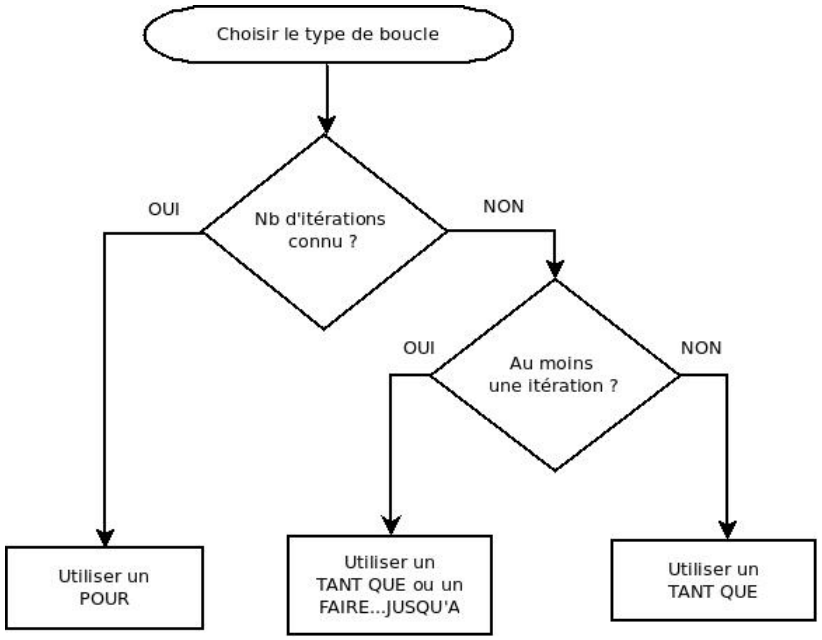
Quel type de boucle choisir ?

En pratique, il est possible d'utiliser systématiquement la boucle `TantQue` qui peut s'adapter à toutes les situations.

Cependant, il est plus clair d'utiliser la boucle `Pour` dans les cas où le nombre d'itérations est fixé et connu à l'avance (par là, on veut dire que le nombre d'itérations est déterminé au moment où on arrive à la boucle).

Quant à la boucle `Répéter`, elle convient dans les cas où le contenu de la boucle doit être parcouru au moins une fois, alors que dans `TantQue`, le nombre de parcours peut être nul si la condition initiale est fausse.

Schéma récapitulatif



Trace d'exécution

La trace d'exécution d'un script comprenant une répétitive se fait en conservant les mêmes règles d'exécution vues dans les modules précédents auquel il faut ajouter celle de la répétitive. Par exemple, dans le cas d'un `TantQue`, il évalue la condition puis :

1. Si la condition est vraie, alors il continue et exécute en séquence les instructions contenues dans son bloc d'instructions entre le mot `Faire` et le mot `FinTantQue`. Lorsqu'il rencontre le mot-clé `FinTantQue`, il revient re-tester la condition.
2. Si la condition est fausse, alors il n'exécute pas le bloc d'instructions, mais va directement au mot-clé `FinTantQue` et poursuit l'exécution à l'instruction qui suit le mot `FinTantQue`.

L'exécution d'une instruction `TantQue` peut être infinie, i.e. ne jamais s'arrêter. On dit qu'elle boucle à l'infini. Pour éviter cette anomalie, le bloc d'instructions doit modifier l'environnement de calcul, les variables, pour que lors d'une évaluation de la condition, celle-ci soit évaluée à faux.

3.2 Le Répéter en TantQue

TantQue/Répéter : Les différences

Les structures `TantQue` et `Répéter` diffèrent sur deux points :

1. **TantQue / Jusqu'à :** Le `TantQue` exécute la séquence d'instructions tant et aussi longtemps que la condition **est** satisfaite, tandis que le `Répéter`, c'est tant et aussi longtemps que la condition **n'est pas** satisfaite.
2. **Avant / Après :** Le `TantQue` vérifie la condition **avant** chaque itération, tandis que le `Répéter` effectue le test **après** chaque itération. La tâche est donc toujours exécutée au moins une fois, sans égard à la valeur de la condition. Cette caractéristique est mise en évidence par la position de la condition dans la structure : elle est située à la fin de celle-ci (alors que dans le `TantQue` la condition est située au début de la structure).

((alg)) **Le Répéter en TantQue**

```
Répéter
|  instructions
Jusqu'à condition
```

Est une écriture simplifiée de :

```
instructions
TantQue (Non condition) Faire
|  instructions
FinTantQue
```

TantQue v.s. Répéter

Répétitive TantQue	Répétitive Répéter
Dans les traitements usuels.	Dans les saisies vérifiées.
Le bloc d'instructions peut ne pas être exécutée.	Le bloc d'instructions est exécuté *au moins* une fois.

3.3 Le Pour en TantQue

Le Pour en TantQue

```
Pour j <- jdebut à jfinal Pas jpas Faire
|  instructions
FinPour
```

Est une écriture simplifiée de :

```
j <- jdebut
Si (jpas > 0) Alors
|   TantQue (j <= jfinal) Faire
|   |  instructions
|   |  j <- j + jpas
|   FinTantQue
Sinon Si (jpas < 0) Alors
|   TantQue (j >= jfinal) Faire
|   |  instructions
|   |  j <- j + jpas
|   FinTantQue
FinSi
```

Rappel

La valeur initiale de la boucle est donnée avant le premier tour de boucle. La condition de fin est vérifiée avant chaque tour de boucle. La valeur de la variable de boucle s'incrémente ou se décrémente automatiquement à la fin de chaque tour de boucle. Il faut donc :

1. Placer une initialisation de la variable de boucle avant le `TantQue`
2. Ajouter une condition pour vérifier que la valeur de fin n'est pas atteinte
3. Incrémenter la variable de boucle à la fin des instructions

Boucle Pour

Dans la séquence d'instructions :

- Veillez à **ne pas modifier** une des variables de contrôle `jdebut`, `jfinal` ou `jpas`.
- Il est **fortement déconseillé** de modifier « manuellement » la variable de contrôle : l'instruction `j <- j + jpas` est automatique et implicite à chaque étape de la boucle.

Enfin il est déconseillé d'utiliser la variable de contrôle à la sortie de la boucle `Pour` sans lui affecter une nouvelle valeur (son contenu pouvant varier selon le langage de programmation).

4 Approfondir : Les suites

Les suites peuvent se résoudre selon un même squelette d'algorithme que nous allons expliquer ici. Puisqu'on doit écrire plusieurs nombres et que l'on sait combien, on se tournera naturellement vers une boucle `Pour`.

4.1 En fonction de j

Le cas le plus simple est lorsque le nombre à afficher à l'étape `j` peut être calculé en fonction de `j` seulement. L'algorithme est alors :

```
// Schéma 1 de résolution d'une suite
Pour j <- 1 à n Faire
  |  Afficher ( f ( j ) )
FinPour
```

Exemple
Pour la suite paire, la fonction est $f(j) = 2 * j$:

- A l'étape 1 on affiche 2
- A l'étape 2 on affiche 4
- ...

Ce qui donne :

Programme @[pgaffPairs1.py]

```python
def PGAffPairs1():
    n = int(input("n? "))
    for j in range(1, n+1):
        print((2 * j), " ", sep="", end="")
    print('\n', end="")

PGAffPairs1()
```

4.2 En fonction du nombre précédent

Parfois, il est difficile (voire impossible) de trouver $f(j)$. On suivra alors une autre approche qui revient à calculer le suivant à partir du nombre que l'on vient d'afficher. La structure générale est alors :

```
// Schéma 2 de résolution d'une suite
nombre <- 1ère valeur à afficher
Pour j <- 1 à n Faire
  |  Afficher ( nombre )
  |  nombre <- calculer le nombre suivant
FinPour
```

Exemple
Pour la suite paire, le 1er nombre à afficher est 2 et le suivant se calcule en ajoutant 2 au nombre courant. D'où :

Programme @[pgsuite1.py]

```python
def PGSuite1():
    n = int(input("n? "))
    nombre = 2
    for j in range(2, n+1):
        print(nombre, " ", sep="", end="")
        nombre += 2
    print()

PGSuite1()
```

Exemple d'exécution

```
n? 10
2 4 6 8 10 12 14 16 18 20
```

Dernier passage dans la boucle
Lors du dernier passage dans la boucle, on calcule une valeur qui ne sera pas affichée. Cette petite perte de temps est dommage mais négligeable et permet de garder une structure claire et générale à la solution.

4.3 Avec une mémoire

Dans certains cas, il n'est pas possible de déduire directement le nombre suivant en connaissant juste le nombre précédent. Prenons un exemple un peu plus compliqué pour l'illustrer :

« Écrivez l'algorithme qui affiche les `N` premiers termes
de la suite : 1, 2, 3, 4, 3, 2, 3, 4, 5, 4, 3... »

Si on vient d'écrire, disons un 3, sans information supplémentaire, impossible de connaitre le nombre suivant. Il faut savoir si on est en phase d'avancement ou de recul et combien de pas il reste à faire dans cette direction.

Ajoutons des variables pour retenir l'**état** où on est :

Programme @[pgsuite2.py]

```python
def PGSuite2():
    n = int(input("n? "))
    nombre = 1
    pasRestants = 3
    direction = 1
    for j in range(1, n+1):
        print(nombre, " ", sep="", end="")
        nombre += direction
        pasRestants -= 1
        if pasRestants == 0:
            direction = -direction
            pasRestants = (3 if direction == 1 else 2)
    print()

PGSuite2()
```

Exemple d'exécution

```
n? 20
1 2 3 4 3 2 3 4 5 4 3 4 5 6 5 4 5 6 7 6
```

Conclusion
On obtient un algorithme plus long mais qui respecte toujours le schéma vu dans l'exemple précédent @[Suite des nombres pairs].

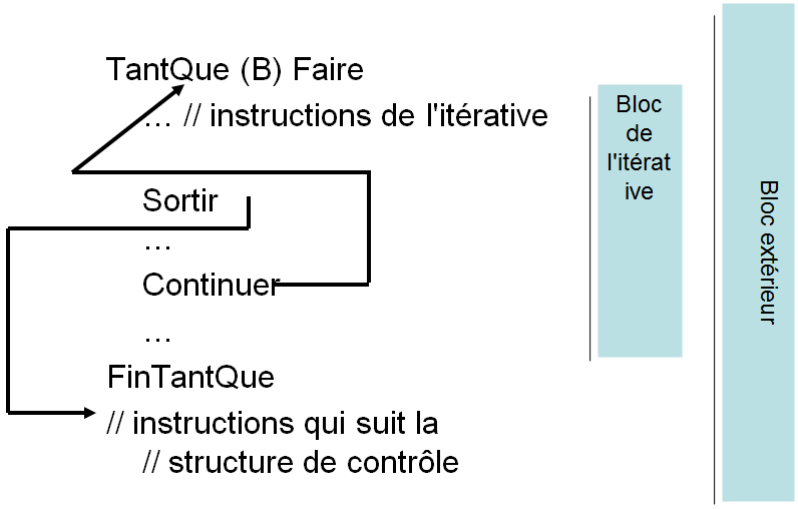
Essayez de respecter ce schéma
Vous obtiendrez plus facilement un algorithme correct et lisible, également dans les cas particuliers.

5 Compléments

5.1 Ruptures de séquence (de bloc)

Ruptures de séquence (de bloc)
On en distingue deux :

- `Sortir` : Interrompt l'exécution de la structure de contrôle en provoquant un saut vers l'instruction qui suit la structure de contrôle.
- `Continuer` : Interrompt l'exécution des instructions du bloc d'une **répétitive** et provoque la ré-évaluation de la condition de continuation afin de déterminer si l'exécution de l'itérative doit être poursuivie (avec une nouvelle itération).



Utilisez-les avec parcinomie
Car elles ne permettent pas de réaliser une preuve formelle d'un algorithme (cf. @[Preuve et Notations asymptotiques]).

Rupture Sortir

```
break
```

Rupture Continuer

```
continue
```