Algorithmique

Chapitre 4: La récursivité

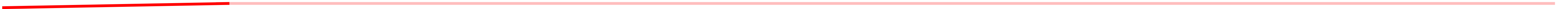
Dr. N'golo KONATE

konatengolo@ufhb.edu.ci

Contenu

1. Exemple introductif
2. Définitions
3. L'intérêt de la récursivité
4. Les principaux types de récursivité
5. Concevoir et écrire une fonction récursive

Exemple introductif



Exemple introductif

Calculer la factorielle d’un entier naturel

Exemple introductif

Calculer la factorielle d'un entier naturel

En mathématiques, il existe deux méthodes pour calculer la factorielle d'un entier naturel n.

Exemple introductif

Calculer la factorielle d'un entier naturel

En mathématiques, il existe deux méthodes pour calculer la factorielle d'un entier naturel n.

- Première méthode : selon la formule classique (cf. définition de la factorielle)

  Si $n = 0$ ou $n = 1$, alors $n! = 1$

  Sinon $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1 = 1 \times 2 \times \ldots \times (n - 2) \times (n - 1) \times n$

Exemple introductif

Calculer la factorielle d'un entier naturel

En mathématiques, il existe deux méthodes pour calculer la factorielle d'un entier naturel n.

- Première méthode : selon la formule classique (cf. définition de la factorielle)

  Si $n = 0$ ou $n = 1$, alors $n! = 1$

  Sinon $n! = n \times (n - 1) \times (n - 2) \times \ldots \times 2 \times 1 = 1 \times 2 \times \ldots \times (n - 2) \times (n - 1) \times n$

  Les algorithmes mettant en œuvre cette méthode utilisent des **boucles**. Ce sont des **algorithmes itératifs**.

Exemple introductif

Calculer la factorielle d'un entier naturel

En mathématiques, il existe deux méthodes pour calculer la factorielle d'un entier naturel n.

- Première méthode : selon la formule classique (cf. définition de la factorielle)

  Si n = 0 ou n = 1, alors n! = 1

  Sinon $n! = n \times (n -1) \times (n -2) \times \ldots \times 2 \times 1 = 1 \times 2 \times \ldots \times (n - 2) \times (n - 1) \times n$

  Les algorithmes mettant en œuvre cette méthode utilisent des **boucles**. Ce sont des **algorithmes itératifs**.

- Deuxième méthode : selon une formule de récurrence

  On démontre à partir de la formule précédente que :

  Si n = 0 alors n! = 1

  Sinon $n! = n \times (n - 1)!$

  Or : $(n-1)! = (n-1) \times (n-2)!$ avec $(n-2)! = (n-2) \times (n-3)!$ et ainsi de suite jusqu'à 0!

Exemple introductif

En conclusion:

Calculer la factorielle d'un entier naturel revient à multiplier cet entier naturel par la factorielle de son précédent : on utilise une factorielle pour calculer une factorielle.

Exemple introductif

En conclusion:

Calculer la factorielle d'un entier naturel revient à multiplier cet entier naturel par la factorielle de son précédent : on utilise une factorielle pour calculer une factorielle.

Ainsi : $5! = 5 \times 4!$

$= 5 \times (4 \times 3!)$

$= 5 \times [4 \times (3 \times 2!)]$

$= 5 \times [4 \times [3 \times (2 \times 1!)]]$

$= 5 \times [4 \times [3 \times [2 \times (1 \times 0!)]]]$

$= 5 \times [4 \times [3 \times [2 \times (1 \times 1)]]]$

$= 5 \times 4 \times 3 \times 2 \times 1$

$5! = 120$

Exemple introductif

En conclusion:

Calculer la factorielle d'un entier naturel revient à multiplier cet entier naturel par la factorielle de son précédent : on utilise une factorielle pour calculer une factorielle.

Ainsi : $5! = 5 \times 4!$

$= 5 \times (4 \times 3!)$

$= 5 \times [4 \times (3 \times 2!)]$

$= 5 \times [4 \times [3 \times (2 \times 1!)]]$

$= 5 \times [4 \times [3 \times [2 \times (1 \times 0!)]]]$

$= 5 \times [4 \times [3 \times [2 \times (1 \times 1)]]]$

$= 5 \times 4 \times 3 \times 2 \times 1$

$5! = 120$

Définitions

Définition

La **récursivité** est la propriété pour un algorithme de s'appeler lui-même un nombre fini de fois. Un algorithme ayant une telle propriété est un **algorithme récursif.**

Définition

La **récursivité** est la propriété pour un algorithme de s'appeler lui-même un nombre fini de fois. Un algorithme ayant une telle propriété est un **algorithme récursif.**

Un algorithme récursif est algorithme A qui s'appelle lui-même ou qui appelle un autre algorithme A' contenant un appel de A.

L’intérêt de la récursivité

L’intérêt de la récursivité

La récursivité est un puissant outil algorithmique (et mathématique où elle est désignée par le terme récurrence) qui permet de décomposer un problème en plusieurs problèmes de même nature mais sur des données plus petites.

L'intérêt de la récursivité

La récursivité est un puissant outil algorithmique (et mathématique où elle est désignée par le terme récurrence) qui permet de décomposer un problème en plusieurs problèmes de même nature mais sur des données plus petites.

Elle est très adaptée à la résolution de certains problèmes n'appartenant pas forcément au domaine des mathématiques (par exemples : la recherche d'un élément dans un tableau trié, le jeu des Tours de Hanoï).

L'intérêt de la récursivité

Propriétés des algorithmes récursifs

Un algorithme récursif a deux propriétés fondamentales :

- il possède des conditions d'arrêt des appels ;
- chaque appel direct ou indirect le rapproche de ses conditions d'arrêt.

Les types de récursivité

L'intérêt de la récursivité

La récursivité directe (ou récursivité simple)

Elle a lieu quand un algorithme s'appelle lui-même.

Exemple : calcul de la factorielle d'un nombre

L’intérêt de la récursivité

la récursivité multiple

Elle a lieu quand un algorithme s’appelle lui-même au moins deux fois simultanément.

Exemple : calcul d’un terme de la suite de Fibonacci

L'intérêt de la récursivité

La récursivité imbriquée

Elle a lieu quand un algorithme appelé dans un autre algorithme s'appelle lui-même (donc elle consiste à effectuer un appel récursif à l'intérieur d'un autre appel récursif).

Exemple : calcul d'une valeur de la fonction d'Ackermann

L’intérêt de la récursivité

La récursivité croisée (ou récursivité indirecte ou récursivité mutuelle)

Elle a lieu quand un algorithme est appelé via une série d’appels d’algorithmes qu’il a lui-même initiés. (Cas simple : l’algorithme A1 appelle l’algorithme A2,

Exemple : détermination de la parité d’un nombre.

L'intérêt de la récursivité

La récursivité croisée (ou récursivité indirecte ou récursivité mutuelle)

Elle a lieu quand un algorithme est appelé via une série d'appels d'algorithmes qu'il a lui-même initiés. (Cas simple : l'algorithme A1 appelle l'algorithme A2,

Exemple : détermination de la parité d'un nombre.

Remarque: Le terme algorithme *récursif* fait référence aux *sous-algorithmes récursifs*, c'est-à-dire aux *procédures récursives* et *fonctions récursives*. Toutefois, dans la pratique, les *fonctions récursives* sont plus utilisées que les procédures récursives.

Concevoir et écrire une fonction récursive

Concevoir et écrire une fonction récursive

Écrire une fonction récursive est délicat. Il faut suivre rigoureusement certaines règles, notamment :

Concevoir et écrire une fonction récursive

Écrire une fonction récursive est délicat. Il faut suivre rigoureusement certaines règles, notamment :

1. vérifier qu'un problème est décomposable

   Le problème doit être décomposable en problèmes de même nature sur des données plus petites.

Concevoir et écrire une fonction récursive

Écrire une fonction récursive est délicat. Il faut suivre rigoureusement certaines règles, notamment :

1. Vérifier qu'un problème est décomposable

   Le problème doit être décomposable en problèmes de même nature sur des données plus petites.

2. Déterminer la (les) condition(s) d'arrêt des appels

   Il faut déterminer la (ou les) condition d'arrêt (ou cas de base) afin de définir le (les) cas pour lequel la fonction ne s'appelle pas elle-même. Si on ne définit pas la (les) condition(s) d'arrêt, la fonction ne terminera pas, c'est-à-dire que la fonction s'appellera indéfiniment.

   Une condition d'arrêt provoque l'arrêt des appels récursifs.

Concevoir et écrire une fonction récursive

Écrire une fonction récursive est délicat. Il faut suivre rigoureusement certaines règles, notamment :

3. Déterminer la (les) condition(s) de continuité des appels

   Il faut déterminer la (les) condition(s) de continuité (ou cas inductif) afin de définir le (les) cas pour lequel la fonction s’appelle elle-même.
   Une condition de continuité provoque un appel récursif.

Concevoir et écrire une fonction récursive

Écrire une fonction récursive est délicat. Il faut suivre rigoureusement certaines règles, notamment :

3. Déterminer la (les) condition(s) de continuité des appels

   Il faut déterminer la (les) condition(s) de continuité (ou cas inductif) afin de définir le (les) cas pour lequel la fonction s'appelle elle-même.
   Une condition de continuité provoque un appel récursif.

4. Ecrire la fonction récursive

   Dans le corps de la fonction récursive, on écrit l'instruction (ou les instructions) contenant le (les) cas de base avant celle(s) contenant le (les) cas inductif(s).
   Chaque appel récursif doit "se rapprocher" d'un cas de base de sorte à favoriser la terminaison de la fonction.

Concevoir et écrire une fonction récursive

Techniquement, on utilise la fonction que l’on n’a pas encore écrite en supposant qu’elle donne déjà un résultat. Une fonction récursive se compose de deux parties :

Concevoir et écrire une fonction récursive

Techniquement, on utilise la fonction que l'on n'a pas encore écrite en supposant qu'elle donne déjà un résultat. Une fonction récursive se compose de deux parties :

- La (les) condition(s) d'arrêts des appels récursifs : dans cette partie les valeurs à déterminer sont directement connues ;

Concevoir et écrire une fonction récursive

Techniquement, on utilise la fonction que l'on n'a pas encore écrite en supposant qu'elle donne déjà un résultat. Une fonction récursive se compose de deux parties :

- La (les) condition(s) d'arrêts des appels récursifs : dans cette partie les valeurs à déterminer sont directement connues ;
- Un (des) appel(s) récursif(s).

Formalisme d’un fonction récursive

Formalisme d'un fonction récursive

Fonction frecursiv (liste des paramètres formels avec leurs types) : type de la valeur de retour

Début

Si (condition d'arrêt)

Alors retourner (r) // valeur de retour de la condition d'arrêt

Sinon retourner (frecursiv (liste des nouveaux paramètres))

FinSi

Fin

Formalisme d’un fonction récursive

L’instruction contenant l’appel récursif peut se présenter sous deux formes telles que :

- La fonction récursive retourne, sans aucun autre calcul, la valeur obtenue par son appel récursif. On parle alors de **récursivité terminale**.
- La fonction récursive retourne, après un autre calcul, la valeur obtenue par son appel

récursif. On parle alors de **récursivité non terminale**.

Exemple : l’appel récursif de la fonction de calcul de la factorielle d’un nombre.

Activité

✓Exemple d'application 1 : Calcul récursif de la factorielle d'un nombre

Écrire une fonction qui calcule la factorielle d'un entier naturel n selon la formule de récurrence $n! = n \times (n-1)!$ sachant que $0! = 1$.

✓Exemple d'application 2 : Calcul d'un terme de la suite de Fibonacci

Écrire une fonction qui calcule le terme de la suite Fn de Fibonacci pour le nombre n sachant que, n étant un entier naturel :

- si $n = 0$ ou $n = 1$ alors $Fn = n$
- si $n > 1$ alors $Fn = Fn - 1 + Fn - 2$.

Activité

✓Exemple d'application 3 : Calcul d'une valeur de la fonction d'Ackermann

Étant donné les entiers naturels m et n, on définit la fonction d'Ackermann $A(m, n)$ par :

- $A(m, n) = A(m - 1 \,;\, A(m \,;\, n - 1))\ pour\ m > 0\ et\ n > 0$
- $A(0, n) = n + 1\ pour\ n > 0$
- $A(m, 0) = A(m - 1 \,;\, 1)\ pour\ m > 0$

On considère que cette fonction retourne la valeur 1 $pour\ m = 0\ et\ n = 0$.

Écrire une fonction qui calcule une valeur de la fonction d'Ackermann.

Activité

✓Exercice d'application 4 : Tester la parité d'un nombre

On suppose que l'on ne connait que la parité du nombre 0 (0 est un nombre pair). Alors pour tester la parité de tout entier naturel n, on utilise les fonctions pair et impair telles que définies ci-dessous :

$$\text{pair}(n) = \begin{cases} \textit{vrai} & \text{si } n = 0; \\ \text{impair}(n-1) & \text{sinon;} \end{cases} \quad \text{et} \quad \text{impair}(n) = \begin{cases} \textit{faux} & \text{si } n = 0; \\ \text{pair}(n-1) & \text{sinon.} \end{cases}$$

Écrire un algorithme qui :

- D'abord reçoit un entier naturel n ;
- Ensuite affiche l'affirmation "n est pair" ;
- Et enfin affiche "VRAI" ou "FAUX" en réponse à l'affirmation précédente.