Récursivité et Paradigme Diviser pour régner

Licence Informatique 2ème année - Ph. Hunel 2025-2026

1

La récursivité et le paradigme « diviser pour régner »

- **Récursivité**

  De l'art d'écrire des programmes qui résolvent des problèmes que l'on ne sait pas résoudre soi-même !

- **Définition (Définition récursive, algorithme récursif).**
  - Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir.
  - Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.
- **Récursivité simple**

  Soit la fonction puissance $x \rightarrow x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & \text{si } n = 0; \\ x \times x^{n-1} & \text{si } n \geq 1. \end{cases}$$

2

La récursivité et le paradigme « diviser pour régner »

- **Récursivité multiple**

  Une définition récursive peut contenir plus d'un appel récursif.

  Calcul des combinaisons C^n_p en se servant de la relation de Pascal :

$$C^p_n = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C^p_{n-1} + C^{p-1}_{n-1} & \text{sinon.} \end{cases}$$

- **Récursivité mutuelle**

  Des définitions sont dites *mutuellement récursives* si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

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

3

La récursivité et le paradigme « diviser pour régner »

- **Récursivité imbriquée**

  La fonction d'Ackermann est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & \text{si } m = 0 \\ A(m-1,1) & \text{si } m > 0 \text{ et } n = 0 \\ A(m-1,A(m,n-1)) & \text{sinon} \end{cases}$$

4

La récursivité et le paradigme « diviser pour régner »

- **Principe et dangers de la récursivité**
  - **Principe et intérêt :** ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :
    - un certain nombre de cas dont la résolution est connue, ces « cas simples » formeront les cas d'arrêt de la récursion ;
    - un moyen de se ramener d'un cas « compliqué » à un cas « plus simple ».
  - La récursivité permet d'écrire des algorithmes concis et élégants.

5

La récursivité et le paradigme « diviser pour régner »

- **Difficultés :**
  - la définition peut être dénuée de sens :

    Algorithme A(n)

    renvoyer A(n)
  - il faut être sûrs que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ;
  - il faut s'assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

6

La récursivité et le paradigme « diviser pour régner »

- **Non décidabilité de la terminaison**

  Question : peut-on écrire un programme qui vérifie automatiquement si un programme donné *P* termine quand il est exécuté sur un jeu de données *D*?

  - **Entrée:** Un programme P et un jeu de données D.
  - **Sortie:** *vrai* si le programme *P* termine sur le jeu de données *D*, et *faux* sinon.
- **Démonstration de la non décidabilité**
  - Supposons qu'il existe un tel programme, nommé ***termine***, de vérification de la terminaison. A partir de ce programme on conçoit le programme Q suivant :

7

La récursivité et le paradigme « diviser pour régner »

```
programme Q
    résultat = termine(Q)
    tant que résultat = vrai faire
        attendre une seconde
    fin tant que
    renvoyer résultat
```

- Supposons que le programme Q —qui ne prend pas d'arguments— termine.
- Donc termine(Q) renvoie *vrai*, la deuxième instruction de Q boucle indéfiniment et Q ne termine pas.
- Il y a donc contradiction et le programme Q ne termine pas.
- Donc, termine(Q) renvoie *faux*, la deuxième instruction de Q ne boucle pas, et le programme Q termine normalement.
- Il y a une nouvelle fois contradiction : par conséquent, il n'existe pas de programme tel que termine.

- **Le problème de la terminaison est indécidable**!!

8

La récursivité et le paradigme « diviser pour régner »

- **Diviser pour régner**
  - **Principe**
    - De nombreux algorithmes ont une structure récursive :

      pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres,

      résolvent les sous problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.
    - Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :
      - **Diviser :** le problème en un certain nombre de sous-problèmes ;
      - **Régner :** sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;
      - **Combiner :** les solutions des sous-problèmes en une solution complète du problème initial.

9

Dérécursivation

Dérécursiver, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

- **Récursivité terminale**

  Définition : *Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.*

10

Récursivité terminale

Exemple :

ALGORITHME **P**(U)

si *C(U)*

alors

D(U);**P**(a(*U*))

sinon *T(U)*

où :

– *U* est la liste des paramètres ;

– *C(U)* est une condition portant sur *U* ;

– *D(U)* est le traitement de base de l'algorithme (dépendant de *U*) ;

– a(*U*) représente la transformation des paramètres ;

– *T(U)* est le traitement de terminaison (dépendant de *U*).

11

Récursivité terminale

- Avec ces notations, l'algorithme P équivaut à l'algorithme suivant :

ALGORITHME **P'**(*U*)

tant que *C(U)* **faire**

D(U);

U ← a(*U*)

fintantque

T(U)

- L'algorithme **P'** est une version **dérécursivée** de l'algorithme **P**.

12

Récursivité non terminale

- Ici, pour pouvoir dérécursiver, il va falloir sauvegarder le contexte de l'appel récursif, typiquement les paramètres de l'appel engendrant l'appel récursif.
- Originellement, l'algorithme est :

ALGORITHME **Q**(U)

si C(U) **alors**

D(U);**Q**(a(U));F(U)

sinon T(U)

13

Récursivité non terminale

⇒Utilisation de **piles**

- Après dérécursivation on obtiendra donc :

```
ALGORITHME Q'(U)
    empiler(nouvel_appel, U)
    tant que pile non vide faire
      dépiler(état, V)
      si état = nouvel_appel alors  U ← V
                                    si C(U) alors D(U)
                                    empiler(fin, U)
                                    empiler(nouvel_appel, a(U))
                                    sinon T(U)
      si état = fin alors U ← V
                          F(U)
```


14

Illustration de la dérécursivation de l'algorithme Q

- Exemple d'exécution de Q :

Appel Q(*U0*)
C(*U0*) *vrai*
D(*U0*)
Appel Q(a(*U0*))
C(a(*U0*)) *vrai*
D(a(*U0*))
Appel Q(a(a(*U0*)))
C(a(a(*U0*))) *faux*
T(a(a(*U0*)))
F(a(*U0*))
F(*U0*)

15

Exemple d'exécution de l'algorithme dérécursivé

- Appel Q'(*U0*)

empiler(nouvel_appel, U))
pile = [(nouvel_appel, *U0*)]
dépiler(état, V))
état ← nouvel_appel ; *V* ← *U0* ; pile = []
U ← *U0*
C(U0) ***vrai***
D(U0)
empiler(fin, U))
pile = [(fin, *U0*)]
empiler(nouvel_appel, a(U)))
pile = [(fin, *U0*) ; (nouvel_appel, a(*U0*))]

16

Exemple d'exécution de l'algorithme dérécursivé

dépiler(état, V))
état ← nouvel_appel ; *V* ← a(*U*0) ; pile = [(fin, *U*0)]
  U ← a(*U*0)
  C(a(**U0**)) ***vrai***
  D(a(**U0**))
  empiler(fin, U))
  pile = [(fin, *U*0) ; (fin, a(*U*0))]
  empiler(nouvel_appel, a(U)))
  pile = [(fin, *U*0) ; (fin, a(*U*0)) ; (nouvel_appel, a(a(*U*0)))]

17

Exemple d'exécution de l'algorithme dérécursivé

dépiler(état, V))
état ← nouvel_appel ; *V* ← a(a(*U*0)) ; pile = [(fin, *U*0) ; (fin, a(*U*0))]
  U ← a(a(*U*0))
  C(a(a(**U0**))) ***faux***
  T(a(a(**U0**)))
dépiler(état, V))
état fin ; *V* ← a(*U*0) ; pile = [(fin, *U*0)]
  F(a(**U0**))
dépiler(état, V))
état ← fin ; *V* ← *U*0 ; pile = []
  F(**U0**)

18

Dérécursivation

- **Remarques**
  - Les programmes itératifs sont souvent plus efficaces,
  - mais les programmes récursifs sont plus faciles à écrire.
  - Les compilateurs savent, la plupart du temps, reconnaître les appels récursifs terminaux, et ceux-ci n'engendrent pas de surcoût par rapport à la version itérative du même programme.
  - Il est toujours possible de dérécursiver un algorithme récursif.

Exercices

```
def f(n):
    if n==0:
        return 1
    else:
        return f(n+1)

def sommebis(n):
    if n==0:
        return 0
    else :
        resultat=sommebis(n-1)
        resultat+=n
        return resultat

def g(n):
    if n<=1:
        return 1
    else:
        return 1+g(n-2)
```

Exercices

- Un nombre est pair si (N−1) est impair, et un nombre N est impair si (N−1) est pair.
  Ecrire deux fonctions récursives mutuelles pair(N) et impair(N) permettant de savoir si un nombre N est pair et si un nombre N est impair.