

1

III. Récursivité

Principe et exemples

Le paradigme « divide & conquer »

Récurtivité

2

- Algorithme récursif vs. itératif
 - ▣ Qui s'appelle lui-même vs. non
- Lien filial avec les maths
 - ▣ Définition du terme d'une suite par une récurrence
- Avantage
 - ▣ Une conception plus simple
 - moins de risque d'effets de bord
- Inconvénient
 - ▣ Une complexité plus importante (pile d'appels)
 - plus lent

Exemples

3

- la factorielle
- La puissance (exponentiation)
- Le pgcd cf. Euclide

La factorielle

4

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{si } n > 0 \end{cases}$$

Algorithme 39: *Factorielle*(n : entier naturel) : entier naturel (version récursive)

Donnée : L'entier naturel n dont on calcule la factorielle

Résultat : $n!$

début

 // Condition de terminaison

si $n = 0$ **alors retourner** 1

 // Propagation récursive

sinon retourner $n \times \text{Factorielle}(n - 1)$

fin

La factorielle

5

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ 1 \times \dots \times n & \text{si } n > 0 \end{cases}$$

Algorithme 41: *Factorielle*(n : entier naturel) : entier naturel (itératif)

Donnée : L'entier naturel n dont on calcule la factorielle
Variable locale : La variable *res* pour mémoriser les résultats intermédiaires
Résultat : La valeur finale $n!$ de *res*

début
 res ← 1
 tant que $n > 0$ **faire**
 res ← *res* × n
 n ← $n - 1$
 fin
 retourner *res*
fin

La puissance

6

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a \times a^{n-1} & \text{si } n > 0 \end{cases}$$

Algorithme 42: *Puissance*(a : nombre, n : entier naturel) : nombre (récuratif)

Donnée : Le nombre a dont on calcule la puissance

Donnée : L'exposant entier n auquel est élevé a

Résultat : a^n

début

 si $n = 0$ alors retourner 1

 sinon retourner $a \times \text{Puissance}(a, n - 1)$

fin

La puissance

7

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ \underbrace{a \times \dots \times a}_n & \text{si } n > 0 \end{cases}$$

Algorithme 43: *Puissance*(a : nombre, n : entier naturel) : nombre (itératif)

Donnée : Le nombre a dont on calcule la puissance
Donnée : L'exposant entier n auquel est élevé a
Variable locale : La variable res pour mémoriser les résultats intermédiaires
Résultat : La valeur finale a^n de res
début
 $res \leftarrow 1$
 tant que $n > 0$ **faire**
 $res \leftarrow res \times a$
 $n \leftarrow n - 1$
 fin
 retourner res
fin

Euclide

8

$$a \wedge b = \begin{cases} a & \text{si } b = 0 \\ b \wedge \left\lfloor \frac{a}{b} \right\rfloor & \text{si } b > 0 \end{cases}$$

Algorithme 1: Algorithme d'Euclide $PGCD(a, b)$

Données : a et b deux entiers naturels

Résultat : le $PGCD$ de a et b

début

 si $b = 0$ alors retourner a

 sinon retourner $PGCD(b, a \bmod b)$

fin

Euclide

9

Algorithme 44: Algorithme d'Euclide $PGCD(a, b)$ (itératif)

Donnée : a et b deux entiers naturels

Variable locale : Reste r des divisions euclidiennes successives

Résultat : le $PGCD$ de a et b

début

$res \leftarrow 1$

tant que $b > 0$ **faire**

$r \leftarrow a \bmod b$

$a \leftarrow b$

$b \leftarrow r$

fin

 retourner a

fin

Mais...

10

- Cf. Euclide, il n'est pas toujours simple de transposer
- Exemple : donner une version itérative de :

Algorithme 45: *NombreElements*(*a* : arbre) : entier (récursif)

Donnée : L'arbre *a* dont on compte les éléments

Résultat : le nombre d'éléments de *a*

début

si *EstArbreVide*(*a*) **alors retourner** 0

sinon retourner 1 + *NombreElements*(*a* → *sag*) + *NombreElements*(*a* → *sad*)

fin

Mais...

11

Algorithme 46: *NombreElements*($t : \text{arbre_binaire}$) : entier (itératif avec pile)

Donnée : L'arbre t dont on compte les éléments
Variable locale : La pile auxiliaire p
Variable locale : La variable n pour compter les éléments
Résultat : le nombre n d'éléments de t

début

si $t = \emptyset$ alors retourner 0

$n \leftarrow 0$

InitialiserPile(p)

Empiler(p, t)

tant que non *EstPileVide*(p) **faire**

$n \leftarrow n + 1$

$t \leftarrow \text{Dépiler}(p)$

 si $t \rightarrow \text{sag} \neq \emptyset$ alors *Empiler*($p, t \rightarrow \text{sag}$)

 si $t \rightarrow \text{sad} \neq \emptyset$ alors *Empiler*($p, t \rightarrow \text{sad}$)

fin

retourner n

fin

Diviser pour régner

12

- Principe
 - ▣ Scinder un problème en sous-problèmes
 - De tailles équivalentes
 - De même nature que le problème principal
 - ▣ Sous-traiter puis agréger les résultats
- Paradigme de conception
 - ▣ Récursif + équilibrage
 - ▣ A la base de nombreux algorithmes puissants
 - En général $O(...n) \rightarrow O(...\ln n)$
- S'adapte très bien à un traitement distribué (//)

Exemples notables

13

- Tris en $O(n \ln n)$
 - ▣ Tri rapide, tri fusion, ...
- Multiplication $O(n^{\text{vous le savez déjà}})$
 - ▣ Karatsuba et ses successeurs
- Compilation
 - ▣ Analyse descendante
- FFT, $O(n \ln n)$
 - ▣ L'algorithme qui en accélère un tas d'autres
 - ▣ Imaginé par Gauss (1805)
 - ▣ Redécouvert par Cooley-Tukey (1965)

Exo: dichotomie

14

- EN : Binary search algorithm
- But : identifier une solution dans un espace de recherche
- Stratégie
 - ▣ Diviser l'espace en deux
 - ▣ Décider dans quelle moitié d'espace chercher
 - ▣ Appliquer récursivement
 - ▣ Arrêter quand l'espace se réduit à un seul élément
- Complexité : $O(\ln n)$
- Application
 - ▣ Recherche de racines de fonctions (théorème des valeurs intermédiaires)
- Exercice
 - ▣ Concevoir un algorithme pour rechercher l'index, dans un tableau trié d'éléments 2 à 2 distincts, d'un élément donné
 - ▣ (et pour retourner 0 si cet élément n'est pas trouvé)

Exo: dichotomie

15

Algorithme 48: *Recherche*($tab : T[n], elt : T, min : index, max : index$) : *index*

Donnée : Un tableau tab strictement ordonné de dimension n
Donnée : Un élément elt dont on cherche l'index dans tab
Donnée : Index min : début du sous tableau de tab dans lequel on recherche
Donnée : L'index max : fin du sous tableau de tab dans lequel on recherche
Variable locale : L'index med : index médian entre min et max
Résultat : L'index recherché de elt dans tab ou 0 si $elt \notin tab$
début
 si $min > max$ **alors retourner** 0
 $med \leftarrow min + (max - min)/2$
 si $tab[med] > elt$ **alors retourner** *Recherche*($tab, elt, min, med - 1$)
 sinon si $tab[med] < elt$ **alors retourner** *Recherche*($tab, elt, med + 1, max$)
 sinon retourner med
fin

Exo: exponentiation rapide

16

$$a^n = \begin{cases} 1 & \text{si } n = 0 \\ a \times a^{n-1} & \text{si } n > 0 \end{cases} \longrightarrow a^n = \begin{cases} 1 & \text{si } n = 0 \\ a \times a^{n-1} & \text{si } n > 0 \text{ et } n \equiv 1(2) \\ \left(a^{\frac{n}{2}}\right)^2 & \text{si } n > 0 \text{ et } n \equiv 0(2) \end{cases}$$

Exo: exponentiation rapide

17

Algorithme 47: *Puissance*(a : nombre, n : entier naturel) : nombre (diviser pour régner)

Donnée : Le nombre a dont on calcule la puissance

Donnée : L'exposant entier n auquel est élevé a

Variable locale : Le résultat res intermédiaire élevé au carré

Résultat : a^n

début

 si $n = 0$ alors retourner 1

 sinon

 si $n \equiv 1(2)$ alors retourner $a \times \text{Puissance}(a, n - 1)$

 sinon

$res \leftarrow \text{Puissance}(a, n/2)$

 retourner $res \times res$

 fin

 fin

fin

$O(n) \rightarrow O(\ln n)$

1 000 000 000 \rightarrow 21

Sauriez-vous le démontrer ?

Exo: multiplication de Karatsuba

18

- Multiplication de deux nombres en numération de position

- ▣ Nombres de $2n$ chiffres

- ▣ Décomposition en deux moitiés

$$\begin{cases} A = A_M b^n + A_m \\ B = B_M b^n + B_m \end{cases}$$

- Représentation algébrique de la multiplication naïve

$$AB = (A_M b^n + A_m)(B_M b^n + B_m) = A_M B_M b^{2n} + (A_M B_m + A_m B_M) b^n + A_m B_m$$

- ▣ Les multiplications sont plus coûteuses que les additions

- ▣ Version naïve : 4 multiplications intermédiaires

- Forme équivalente de Karatsuba

- ▣ Complexification $AB = A_M B_M b^{2n} + ((A_M + A_m)(B_M + B_m) - (A_M B_M + A_m B_m)) b^n + A_m B_m$

- Mémorisation et réutilisation de résultats intermédiaires

- Ajout de 3 opérations additives supplémentaires

- ▣ On tombe à 3 multiplications !

- Application **récursive** du procédé : $O(n^2) \rightarrow O(n^{1,58})$

Exo: multiplication de Karatsuba

19

- Nous disposons d'utilitaires
 - ▣ *longueur(N)* : retourne le nombre de chiffres de N
 - ▣ *droite(N, k)* : supprime k chiffres sur la droite de N
 - ▣ *gauche(N, k)* : ajoute k de zéros sur la droite d'un Nombre
 - ▣ *produit_scalaire(N, s)* : simplement $N + \dots + N$, s fois
- Principe
 - ▣ Si $A < B$, interchanger les opérandes
 - ▣ Mesurer les longueur l_A et l_B
 - Si B est un scalaire, terminaison : retourner le *produit_scalaire*
 - Sinon scinder A et B et produire les termes dérivés
 - Effectuer les 3 produits par délégation (appels récurifs)
 - Assembler et retourner le résultat

Exo: multiplication de Karatsuba

20

Algorithme 49: *Multiplication(A : entier, B : entier) : entier*

Donnée : Le premier opérande A
Donnée : Le second opérande B
Variable locale : La longueur l_A de l'écriture du premier opérande A
Variable locale : La position k du point de coupure des opérandes
Variable locale : $A_M, A_m, B_M, B_m, A_{Mm}, B_{Mm}$: termes dérivés
Variable locale : $AB_{MM}, AB_{Mm}, AB_{mm}$: les trois produits intermédiaires
Résultat : Le produit $A \times B$
début
 si $B > A$ **alors** $A \leftrightarrow B$ // On force la convention $A \geq B$
 // Cas d'un simple produit scalaire
 si $\text{longueur}(B) < 2$ **alors retourner** $\text{produit_scalaire}(A, B)$
 // Sinon, on repère le point de coupe en fonction de l_A
 $l_A \leftarrow \text{longueur}(A)$
 $k \leftarrow \lceil \frac{l_A}{2} \rceil + \lfloor \frac{l_A}{2} \rfloor$ // Noter la petite optimisation (facultative)

Exo: multiplication de Karatsuba

21

```
// Extraction des 4 moitiés d'opérandes (coût négligeable)
 $A_M \leftarrow droite(A, k)$ 
 $A_m \leftarrow A - gauche(A_M, k)$ 
 $B_M \leftarrow droite(B, k)$ 
 $B_m \leftarrow B - gauche(B_M, k)$ 
// Production des termes dérivés (coût négligeable)
 $A_{Mm} \leftarrow A_M + A_m$ 
 $B_{Mm} \leftarrow B_M + B_m$ 
// Les 3 produits intermédiaires
 $AB_{MM} \leftarrow Multiplication(A_M, B_M)$ 
 $AB_{Mm} \leftarrow Multiplication(A_{Mm}, B_{Mm})$ 
 $AB_{mm} \leftarrow Multiplication(A_m, B_m)$ 
// Retour du résultat après assemblage (coût négligeable)
retourner  $gauche(AB_{MM}, 2 \times k) + gauche(AB_{Mm} - AB_{MM} - AB_{mm}, k) + AB_{mm}$ 
```

fin

Pour conclure

22

- Application : changement de base
 - ▣ Conception
 - Cela revient à évaluer un polynôme
 - Concevez la version naïve
 - Concevez une version qui réunit
 - Algorithme de Horner
 - Exponentiation rapide
 - Multiplication de Karatsuba
 - ▣ Comparez !

A retenir

23

- La version itérative est un peu plus performante que la version récursive d'un même algorithme
- Concevoir suivant le paradigme récursif du *diviser pour régner* produira généralement un algorithme radicalement plus performant que celui pensé suivant un paradigme itératif