UCLouvain

Algorithmique

Cours préparatoires au master en sciences informatiques

Vincent Branders

2022–2023

Fonctions récursives et équations de récurrence

Principe de la récursion

Définition

Une fonction est récursive si elle s'appelle elle-même, directement ou indirectement

Avantages

- une manière, compacte et claire, de représenter/concevoir un algorithme
- un mode de pensée très naturel, une fois acquis
- un lien direct avec une preuve par induction pour démontrer l'exactitude d'un algorithme

Un premier exemple : la fonction factorielle

Définition

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

Algorithme itératif :

Algorithm: factorielleIter
Input: Un entier naturel n
Output: $n!$
$fact \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $fact \leftarrow fact \times i$
return $fact$

Approche récursive :

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

Algorithme récursif :

Algorithm: factorielle
Input: Un entier naturel n
Output: $n!$
if $n == 0$ **then** // Cas de base
  return 1
return $n \times$ `factorielle(n-1)` // Appel récursif

Schéma général de la récursion

Algorithm: factorielle
Input: Un entier naturel n
Output: $n!$
if $n == 0$ **then** `// Cas de base`
  return 1
return $n \times$ `factorielle(`n-1`)` `// Appel récursif`

En général

1. Définir un ou plusieurs **cas de base**
2. Construire un ou plusieurs **appel(s) récursif(s)** sur un ou des **sous-problème(s)**
3. Combiner correctement le **résultat des appels récursifs**

Questions à approfondir

Algorithm: factorielle
Input: Un entier naturel n
Output: $n!$
if $n == 0$ **then** // Cas de base
  return 1
return $n \times$ `factorielle(n-1)` // Appel récursif

- le gain en clarté n'apporte pas nécessairement un gain en complexité temporelle
  - ici, en itératif ou en récursif : $\Theta(n)$
  - l'analyse rigoureuse de la complexité temporelle d'une méthode récursive nécessite la résolution d'un système d'équations de récurrence (voir plus tard)
- contrôle de flux : où revient-on après l'instruction return ?
- pièges à éviter lors de l'écriture d'une méthode récursive

Erreurs classiques

- La précondition n'est pas vérifiée ($n < 0$) : récursion sans fin

```
x ← factorial(-1)
```

- Oublier le(s) cas de base

```
return n × factorial(n-1)
```

- Mal positionner le(s) cas de base

```
return n × factorial(n-1)
if n = 0 then
    return 1
```

- Appel récursif mais sans **return** pour récupérer le résultat

```
if n = 0 then
    return 1
n × factorial(n-1)
```

- Appel récursif sans réduction à un sous-problème

```
if n = 0 then
    return 1
return n × factorial(n)
```

La récursion résumée

- La récursion offre une manière de programmer compacte et claire
- La récursion nécessite :
  - un ou plusieurs cas de base
  - un ou plusieurs appels récursifs sur des sous-problèmes
  - de récupérer correctement la valeur renvoyée par les appels récursifs
- La récursion terminale offre un gain en complexité spatiale, lorsque le compilateur/interpréteur en tire parti

Exercices sur la récursion (si nécessaire) :

https://inginious.org/course/prepa-master/Algo_14_Expo_rec

https://inginious.org/course/prepa-master/Algo_15_Max_rec

https://inginious.org/course/prepa-master/Algo_16_Size_rec

https://inginious.org/course/prepa-master/Algo_17_Sum_rec

Comment analyser la complexité d'algorithme récursif ?
Grâce aux équations de récurrence

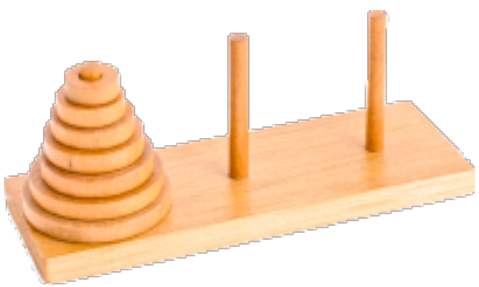
Résolution par substitution

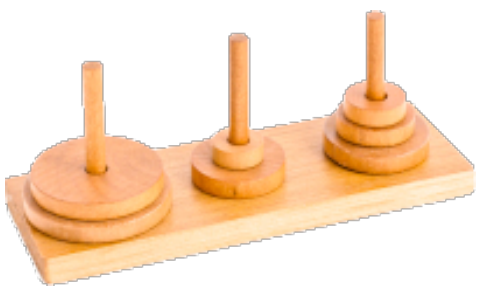
1. Énoncer les équations de récurrence en posant $T(n)$ le temps mis pour résoudre un problème de taille n en fonction
   - de constantes (ex. $c1$, $c2$),
   - de $f(n)$ (ex. : n, n^2, $\log n$, ...)
   - du temps mis pour résoudre un ou plusieurs sous-problèmes (ex. : $T\left(\frac{n}{2}\right)$, $T(n-1)$, ...)
2. Spécifier la taille et la complexité du (ou des) cas de base (souvent une constante b)
3. Substituer le temps pour les sous-problèmes en utilisant la même équation (mais en adaptant la valeur de n)
4. Répéter la substitution, jusqu'à trouver une formule générale en fonction de i (la profondeur de la récursion)
5. Remplacer i par i_{max} (la profondeur maximale de la récursion) dans la formule générale
6. Laisser tomber les constantes et termes non-dominants pour formuler la complexité en $\mathcal{O}$ ou Θ

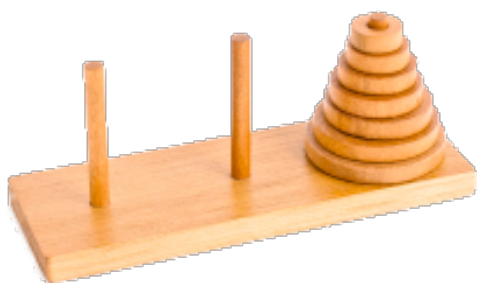
But du jeu

Déplacer tous les disques du piquet gauche au piquet droit

- un seul disque peut être déplacé à chaque mouvement
- un disque peut uniquement être déplacé sur
  - un piquet vide
  - un disque plus grand







Équation de récurrence : exemple

Algorithm: move
Input: Le nombre $n > 0$ de disques à déplacer
Input: Un booléen qui indique si l'on déplace les disques vers *Left* ou non
Output: Affiche les mouvements pour résoudre une tour de Hanoi d'ordre n

```
if n == 0 then
    return
move(n-1, not left)
if left then
    print(n, "left")
else
    print(n, "right")
move(n-1, not left)
```

$$T(0) = b$$
$$T(n) = c + 2T(n-1)$$
$$T(n) = c + 2\left[c + 2T(n-2)\right]$$

Réarranger les termes :
$T(n) = 3c + 4T(n-2)$

Remplacer $T(n-2)$ en utilisant la même équation :
$T(n) = 3c + 4\left[c + 2T(n-3)\right]$

Réarranger les termes :
$T(n) = 7c + 8T(n-3)$

Trouver une forme générale à l'étape i :
$T(n) = \left(2^i - 1\right).c + 2^i T(n-i)$

i représente la profondeur de la récursion, que vaut i_{max} ? $i_{max} = n$
$T(n) = (2^n - 1)\,c + 2^n.T(0) = 2^n.(c+b) - c$

$$\Longrightarrow \quad T(n) \in \Theta\left(2^n\right)$$

Exercices sur la complexité temporelle d'algorithmes récursifs :

https://inginious.org/course/prepa-master/Algo_18_insertionsort

https://inginious.org/course/prepa-master/Algo_19_insertionsort2

https://inginious.org/course/prepa-master/Algo_20_insertionsort3

https://inginious.org/course/prepa-master/Algo_21_mountain