

Récursion, mémoïsation, programmation dynamique

Cours L3 Algorithmique et programmation

Pierre Senellart



12 octobre 2017

Plan

Récursion

Mémoïsation

Programmation dynamique

Quand utiliser la récursion ?

Un problème P peut être résolu par récursion quand :

- Il peut être paramétré par **un ou plusieurs entiers**
 $P(n_1 \dots n_k)$
- La solution de $P(n_1 \dots n_k)$ peut être calculée **à partir de** la solution de $P(n_1^{(1)} \dots n_k^{(1)}) \dots P(n_1^{(\ell)} \dots n_k^{(\ell)})$ pour **un certain $\ell \geq 1$** avec pour tout $1 \leq i \leq \ell$, chaque $n_j^{(i)} \leq n_j$ pour $1 \leq j \leq k$, l'une de ces inégalités étant stricte ($\exists 1 \leq j \leq k, n_j^{(i)} < n_j$) : **cas récursif**
- $P(0 \dots 0)$ (ou $P(n_1 \dots n_k)$ avec n_i « petit » suivant les cas) **facile à calculer** : **cas de base**

Exemple 1 : Factorielle

- $P(n) = n!$
- $k = 1, \ell = 1$
- $P(n) = n \times P(n - 1)$ pour $n \geq 1$
- $P(0) = 1$

Exemple 2 : Fibonacci

- $P(n)$ n -ième nombre de Fibonacci
- $k = 1, \ell = 2$
- $P(n) = P(n - 1) + P(n - 2)$ pour $n \geq 1$
- $P(0) = 1, P(1) = 1$

Exemple 3 : distance d'édition de Levenshtein

$d(s, s')$ distance d'édition (nombre minimal de caractères à ajouter, enlever, modifier) pour passer de s à s' .

- $P(n_1, n_2)$ distance d'édition entre le préfixe de longueur n_1 de s et le préfixe de longueur n_2 de s'
- $k = 2, \ell = 3$
- $P(n_1, n_2) = \min \left(P(n_1 - 1, n_2) + 1, P(n_1, n_2 - 1) + 1, P(n_1 - 1, n_2 - 1) + \mathbb{I}_{s_{n_1} \neq s'_{n_2}} \right)$
pour $n_1 \geq 1, n_2 \geq 1$
- $P(n_1, 0) = n_1$ pour $n_1 \geq 0$; $P(0, n_2) = n_2$ pour $n_2 \geq 0$.

\mathbb{I}_b vaut 1 si b est vrai, 0 sinon

Implémentation

- **Fonction récursive**

```
function  $P(n_1, \dots, n_k)$ ;
```

```
if cas de base then
```

```
    ...;
```

```
    return ...;
```

```
else
```

```
    // cas récursif
```

```
    ...;
```

```
    //  $\ell$  appels à  $P$ 
```

```
    return ...;
```

```
end
```

- On dit que la récursion est terminale si $\ell = 1$ et si l'appel à P est la dernière instruction appelée du cas récursif (**return** $P(n'_1, \dots, n'_k)$)
- NB : Dans l'exemple ci-dessus le **else** peut être omis.

Complexité

- Bornes **supérieures** (parfois meilleures, par exemple si l'appel récursif se fait sur $\frac{n}{2}$ et pas sur $n - 1$) :
 - Si $\ell = 1$: $O(n_1 \times \dots \times n_k)$: souvent **acceptable**
 - Si $\ell > 1$: $O(\ell^{n_1 \times \dots \times n_k})$: souvent **non raisonnable**
- Attention à l'espace mémoire **sur la pile** : $O(\prod_{i=1}^k n_i)$ (en général l'espace allouée à la pile va de quelques centaines de kilo-octets à quelques méga-octets).
- En cas de récursion terminale, le compilateur du langage de programmation peut éliminer la récursion, et donc l'espace sur la pile requis

Plan

Récursion

Mémoïsation

Programmation dynamique

Principe (1/3)

Mémoire : objet global (ou statique, ou membre de classe) qui se souvient des résultats de la fonction P d'un appel à l'autre

Principe (2/3)

```
function  $P(n_1, \dots, n_k)$ ;  
if  $M(n_1, \dots, n_k)$  est défini then  
  | return  $M(n_1, \dots, n_k)$ ;  
end  
if cas de base then  
  | ... ;  
  |  $r \leftarrow \dots$ ;  
else  
  | // cas récursif  
  | ... ;  
  | //  $\ell$  appels à  $P$   
  |  $r \leftarrow \dots$ ;  
end  
 $M(n_1, \dots, n_k) \leftarrow r$ ;  
return  $r$ ;
```

Principe (3/3)

- Récursion terminale impossible !
- Structure de données : **tableau multi-dimensionnel** ou **tableau associatif** (**map**) implémenté comme un arbre binaire de recherche équilibré ou comme une table de hachage (voir cours ultérieur)
- Choisir un tableau classique quand les paramètres sont des entiers à **valeurs contiguës**, choisir un tableau associatif sinon
- Les tableaux peuvent généralement être alloués statiquement quand on dispose d'un **majorant** a priori de la taille des paramètres

Complexité

- $O(n_1 \times \dots \times n_k \times \ell)$ opérations
- $O(n_1 \times \dots \times n_k)$ espace mémoire sur le tas (espace occupé dépend de la structure de données utilisée)
- $O(n_1 \times \dots \times n_k)$ espace mémoire sur la pile
- Pas de calculs inutiles !

Plan

Récursion

Mémoïsation

Programmation dynamique

Principe

- Calcul **itératif**
- On construit une **matrice** (tableau multi-dimensionnel) $n_1 \times \dots \times n_k$ qu'on remplit itérativement du bas vers le haut

```
function  $P(n_1, \dots, n_k)$ ;  
 $M \leftarrow$  matrix( $n_1, \dots, n_k$ );  
// remplir  $M$  avec les cas de base  
for  $i_1 \leftarrow 1$  to  $n_1$  do  
|  
|   ... ;  
|   for  $i_k \leftarrow 1$  to  $n_k$  do  
|   |  
|   |   ... ;  
|   |    $M(i_1, \dots, i_k) \leftarrow \dots$ ;  
|   |   // utilise les valeurs calculées  
|   end  
end  
return  $M(n_1 \dots n_k)$ ;
```

Complexité

- $\Theta(n_1 \times \cdots \times n_k \times \ell)$ opérations
- $\Theta(n_1 \times \cdots \times n_k)$ espace mémoire sur le tas
- $O(1)$ espace mémoire sur la pile
- Parfois trop d'opérations !

Mémoïsation vs Programmation dynamique

- Les deux techniques reviennent **essentiellement au même**
- **Avantage de la mémoïsation** : seuls les calculs nécessaires sont effectués (on ne remplit pas une matrice complète)
- **Inconvénient de la mémoïsation** : les appels récursifs ont un léger surcoût (par rapport à un style impératif) et utilisent la pile d'appel (limitée en taille)
- On peut reproduire le comportement de la programmation dynamique à partir de la mémoïsation
- On peut dans certains cas complexes simuler les appels récursifs avec une pile maintenue à la main sur le tas