Chapitre 4

Récursivité

Notions	Commentaires
Récursivité d'une fonction. Récursivité croisée. Organisation des activations sous forme d'arbre en cas d'appels multiples.	La capacité d'un programme à faire appel à lui-même est un concept primordial en informatique. [...] On se limite à une présentation pratique de la récursivité comme technique de programmation.

Extrait de la section 2 du programme officiel de MP2I : « Récursivité et induction ».

Sommaire

0 Introduction

0.0 Notion de réduction

Rappels du cours introductif : Un **problème** est composé de deux éléments : la description d'une **instance** (une « entrée »), et une question/tâche à réaliser sur cette instance.

Définition 1 (Réduction).
On dit qu'une instance d'un problème A se *réduit* à une instance d'un problème B si résoudre cette instance de B suffit à résoudre cette instance de A.
On dit que le problème A se réduit au problème B si toute instance de A se réduit à une instance de B. On note parfois $A \preccurlyeq B$.

Remarque.

- Dit autrement, il s'agit d'utiliser le fait que l'on sait déjà résoudre un problème B afin de résoudre un problème A. C'est l'une des méthodes courantes en science : se ramener à ce que l'on sait faire !
- Vous approfondirez cette notion en MPI, notamment en imposant des conditions sur les liens entre les instances.
- Notez qu'il n'y a pas besoin de savoir *comment* B est résolu. D'un point de vue pratique, il suffit d'avoir accès à une fonction qui résout B (sans même savoir comment elle marche) afin de résoudre A.

Exemple.

- Posons A le problème de calculer le PGCD de deux entiers (une instance est la donnée de deux entiers), et B le problème de calculer le PPCM de deux entiers.
  Alors A se réduit à B. En effet, soit (x, y) une instance de A. Utilisons (x, y) comme une instance de B. Or, d'après le cours de maths, on sait que $|xy| = pgcd(x, y).ppcm(x, y)$ et donc que $pgcd(x, y) = \frac{|xy|}{ppcm(x,y)}$.
  Ainsi, résoudre l'instance (x, y) de B suffit à calculer $pgcd(x, y)$ et donc à résoudre l'instance (x, y) de A. Il s'ensuit que A se réduit à B.
- On peut montrer que réciproquement, PPCM se réduit à PGCD.

0.1 Notion de récursion

L'idée centrale de la récursion est de réduire une instance d'un problème à une autre instance, plus petite, de ce même problème.

Exemple.

- Le calcul d'une suite récurrente en maths : si par exemple $u_{n+1} = f(u_n)$, alors l'instance « calculer le terme $n + 1$ » se réduit à l'instance « calculer le terme n ».
- Le calcul des coefficients binomiaux : pour calculer $\binom{n}{k}$, d'après le cours de maths il suffit de calculer $\binom{n-1}{k}$ et $\binom{n-1}{k-1}$.
- Dessiner une fractale : une fractale est un dessin qui se répète dans lui-même. Ainsi, pour dessiner une fractale « grande », il faut commencer par dessiner des fractales plus « petites » dedans.

Exercice. Montrez que le problème de trouver le minimum d'un tableau se réduit au problème de trier un tableau.

Cette méthode de résolution de problèmes se généralise à l'écriture des fonctions :

Définition 2 (Fonction récursive).
Une **fonction récursive** est une fonction qui peut faire un ou plusieurs appels à elle-même.
Le contenu d'une fonction récursive est composée de :

- Cas récursif : les appels que la fonction fait à elle-même, et ce qu'elle en fait.
- Cas de base : ce que fait la fonction quand elle ne peut pas s'appeler elle-même.

Exemple.

exemples.c

```c
/** Renvoie x^n. */
double exp_naif(double x, unsigned n) {
  if (n > 0) {
    return x * exp_naif(x, n-1);
  } else { // n == 0
    return 1;
  }
}
```

La fonction ci-contre calcule x^n (avec $n \geq 0$). Pour cela :

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

On peut représenter visuellement le déroulement de cette fonction sur des exemples :

(a) `exp_naif(1.5, 4)` (b) `exp_naif(2.0, 5)`

FIGURE 4.1 – Visualisation d'appels à `exp_naif`

1 Exemples

1.0 Affichage d'un triangle

Écrivons une fonction qui affiche un triangle de n lignes d'étoiles, comme ceci :

```
***
**
*
```

(a) Triangle de n=3 étoiles

```
*****
****
***
**
*
```

(b) Triangle de n = 5

FIGURE 4.2 – Inclusion des triangles

Essayons d'écrire une fonction récursive pour ce problème. Pour cela, cherchons une « structure récursive » dans le problème.

On remarque que le triangle n-1 est inclus dans le triangle n :

```
*****
****
***
**
*
```

FIGURE 4.3 – Inclusion des triangles

On en déduit la solution récursive suivante :

Fonction Triangle

Entrées : $n \geq 0$

```
1 si n > 0 alors
2 |   Afficher une ligne de n étoiles
3 |   Triangle(n-1)
```

Remarque. Pour bien comprendre cette fonction, il faut la faire tourner à la main.

Exercice. Écrire cette fonction en C, et la tester. *Il n'y a aucune syntaxe particulière à utiliser pour la récursivité en C.*

Exercice. Faire de même mais avec un triangle "pointe en haut".

1.1 Tours de Hanoï

Les tours de Hanoï sont un casse-tête très célèbre :

- On dispose de 3 tiges (emplacements), sur lesquelles se trouvent des disques, empilés les uns sur les autres.
- Il y a un total de n disques. Ils sont caractérisés par leur diamètre, qui sont deux à deux distincts. Un disque ne peut jamais reposer sur un disque plus petit.
- On peut déplacer le disque qui est au sommet d'une pile en le plaçant au sommet d'une autre (en respectant la règle des diamètres !).
- Initialement, tous les disques sont sur une même tige. Le but est de tous les déplacer sur une autre des 3 tiges.

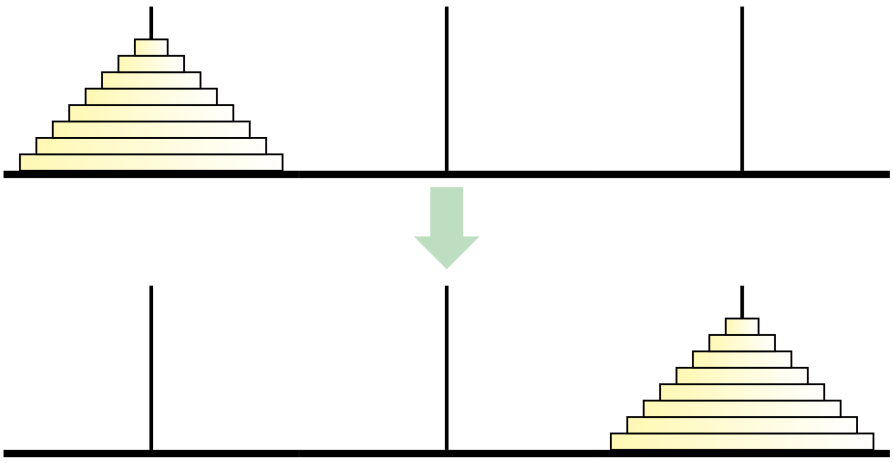

FIGURE 4.4 – Situation initiale et finale des tours de Hanoï avec n=8 disques (src : *Algorithms*, J. Erikson)

Exemple. Voici une succession de déplacements qui résout les tours de Hanoï à 3 disques :

Figure 4.5 – Résolution de Hanoï à 3 disques

On veut écrire une solution récursive au problème : on veut trouver une méthode pour déplacer n disques d'une tige vers une autre. Pour cela, **on suppose que l'on a des camarades très intelligent·es qui savent déjà le résoudre pour des instances plus petites**. L'objectif est de réussir à utiliser leur aide pour résoudre notre instance.

En bidouillant un peu, on réalise qu'une étape intermédiaire inévitable dans toute solution est de placer le plus gros disque (celui du fond de la pile de départ) sur la tige d'arrivée. Il faut donc réussir à enlever les autres disques d'au-dessus de lui, et comme il est plus gros que les autres il faut que la tige d'arrivée soit vide quand on le place. Autrement dit, il faut déplacer les n-1 premiers disques sur la tige qui n'est ni départ ni arrivée.

...[1]

Mais en fait « déplacer n-1 premiers disques d'une tige à une autre », c'est exactement une autre instance du problème ! Nos camarades peuvent donc le résoudre ! Il nous suffit ensuite de déplacer le gros disque sur la tige d'arrivée, puis de re-déplacer les n-1 premiers.

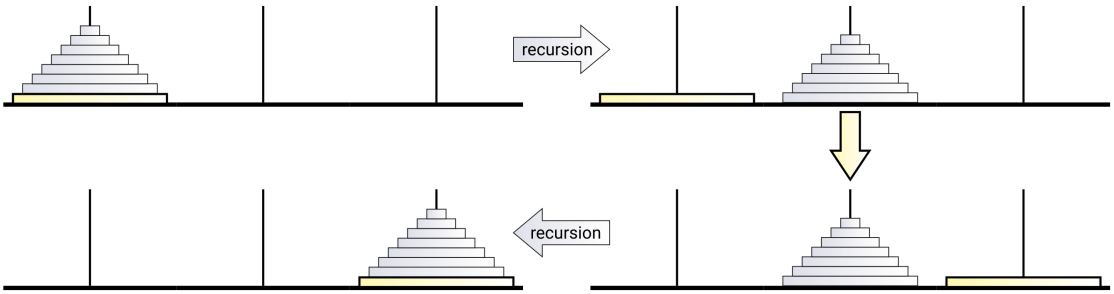

Figure 4.6 – Le fonctionnement récursif de l'algorithme (src : *Algorithms*, J. Erikson)

1. Insérer roulements de tambours dramatiques.

On a arrête de réflechir. On ne cherche SURTOUT PAS à « déplier » les appels récursifs : nos camarades sont très intelligent·es et on leur fait confiance !!!!!!!!!!

Il reste une subtilité : quand n = 0, l'étape « déplacer n -1 disques » n'a pas de sens. Dans ce cas, on ne *peut pas* appliquer notre méthode récursive et il faut résoudre cette instance par nous-même. Heureusement, déplacer n = 0 disques n'est pas trop compliqué... On obtient donc le pseudo-code suivant :

Fonction Hanoï

Entrées : i : la tige de départ ; j : la tige d'arrivée ; n : le nombre de disques à déplacer

```
1 si n > 0 alors
2     k ←tige autre que i ou j
3     Hanoï(i, k, n-1)
4     Déplacer 1 disque de i à j
5     Hanoï(k, j, n-1)
```

Exemple. En fait, la résolution précédente de Hanoï à 3 disques était déjà une application de cet algorithme !

Exercice. Appliquer cet algorithme récursif pour n = 4. *Conseil : prévoyez beaucoup de place, et notez les appels récursifs au fur et à mesure ; votre mémoire ne suffira pas.*

Remarque.

- J'insiste : pour trouver cette solution, nous n'avons **pas** cherché à comprendre comment les camarades intelligent·es procèdent !! Les camarades / appels récursifs sont des « boites noires » qui résolvent les autres instances, sans que l'on ait à comprendre comment.
- De manière cachée, nous avons commencé par **généraliser** : on n'a pas directement résolu le problème de « déplacer n tiges du départ à l'arrivée », on a résolu le problème de « déplacer les n *plus petits* disques d'une tige (où ils doivent tous être) vers une autre ».
- En particulier, ce qui fait marcher cet algorithme est que l'on essaye toujours de déplacer les disques les plus petits : même si on ne le pose pas sur des tiges vides (la tige intermédiaire est rarement vide), on les pose sur des disques plus gros : tout va bien.
- Il est formateur d'essayer d'analyser la complexité de cet algorithme : cf cours complexité.

1.2 Exponentiation rapide

La fonction `exp_naif` précédente est une implémentation de la fonction $(x, n) \mapsto x^n$. Si on compte sa complexité en nombre de multiplications $M(n)$, on obtient l'équation $M(n) = 1+M(n-1)$ et $M(0) = 0$, donc $M(n) = n$.

Cela peut sembler bien, mais c'est en fait assez mauvais[2]. Il existe une méthode plus maline, basée sur la remarque suivante : notons $p = \left\lfloor \frac{n}{2} \right\rfloor$ et utilisons :

$$x^n = \begin{cases} x.x^p.x^p & \text{si } n = 2p+1 \text{ et } n > 0 \text{ , c'est à dire si } n \text{ est impair et } > 0 \\ x^p.x^p & \text{si } n = 2p \text{ et } n > 0 \quad \text{, c'est à dire si } n \text{ est pair et } > 0 \\ 1 & \text{sinon} \quad \text{, c'est à dire si } n = 0 \end{cases}$$

2. Vous en parlerez plus en MPI, mais faisons un brin de hors-programme : la complexité « pertinente » s'exprime en fonction de la taille des entrées. n s'écrit sur $\log_2 n$ bits, donc une complexité en $\Theta(n)$ est exponentielle en la taille de l'entrée. Et l'exponentiel, c'est beaucoup.

On obtient le code suivant :

exemples.c

```c
/** Renvoie x^n. */
double exp_rap(double x, unsigned n) {
  if (n > 0) {
    int x_p = exp_rap(x, n/2); // x puissance p avec p = n/2
    if (n % 2 == 0) { return x_p * x_p; }
    else { return x * x_p * x_p; }
  }
  else {
    return 1;
  }
}
```

Ou en OCaml :

exemples.ml

```ocaml
(** Renvoie x^n (avec n >= 0) *)
let rec exp_rap x n =
  if n > 0 then
    let x_p = exp_rap x (n/2) in (* x puissance p avec p = n/2 *)
    if n mod 2 = 0 then x_p *. x_p else x *. x_p *. x_p
  else
    1.
```

La complexité $M(n)$ de cet algorithme-ci vérifie $M(n) \leq 2 + M(n/2)$ et $M(0) = 0$. On en déduit[3] $M(n) = O(\log_2 n)$, ce qui est *bien* mieux que la complexité précédente !

Remarque.

- On être plus précis sur la complexité de cet algorithme. Notons N_0 le nombre de 0 dans l'écriture binaire de n, et N_1 celui de 1 (on a en particulier $N_0 + N_1 \simeq \log_2 n$).
  Avec ces notations, on peut[4] montrer que $M(n) = N_0 + 2N_1$ qui est bien un $O(\log_2 n)$.
- On peut prouver qu'il faut $\Omega(\log_2 n)$ opérations élémentaires. En effet, on peut prouver qu'il *faut* lire chacun des bits de n.
  Pour cela, raisonnons par l'absurde et supposons qu'il existe un algorithme totalement correct qui fonctionne sans lire tous les bits de l'entrée n. Soit $n \in \mathbb{N}$ tels qu'un bit de n ne soit pas lu. Notons m l'entier obtenu en changeant la valeur de ce bit non lu. Alors l'algorithme renvoie la même réponse sur x^m et x^n... mais ces deux valeurs sont distinctes[5] : l'algorithme n'est donc pas correct, absurde.
  Il faut donc lire chacun des bits de n, et donc effectuer au moins $\Omega(\log_2 n)$ opérations élémentaires. Sachant cela, effectuer $O(\log_2 n)$ multiplications est très satisfaisant.

⚠ Attention, une erreur commune avec cet algorithme est d'effectuer deux appels récursifs à chaque fois (ce que l'on évite dans les codes ci-dessus en stockant le résultat de l'appel dans une variable). Si on fait cela, la complexité redevient[6] $O(n)$.

1.3 À propos des boucles

1.3.0 Transformation boucle <-> récursion

La plupart des boucles peuvent être réécrites par une fonction récursive. L'idée est de faire une fonction qui « fait une itération », puis qui s'appelle elle-même pour faire la prochaine itération. Ainsi, les deux codes ci-dessous sont équivalents :

3. Pour cela, remarquer que $M(1) = 2$ simplifie la manipulation des log.
4. Il faut étudier l'écriture binaire de n durant la suite d'appels récursive, et utiliser le fait que le dernier bit indique la parité
5. Sauf pour $x = 0$ ou $|x| = 1$, d'accord.
6. Utiliser la méthode de calcul de complexité par arbre.

exemples.c

```c

/** Somme des entiers jusqu'à n
    ↪ inclus. */
int somme_entiers(int n) {
  int i = 0;
  int somme = 0;
  while (i <= n) {
    somme = somme + i;
    i = i +1;
  }
  return i;
}
```

exemples.ml

```ml

(** Somme des entiers jusqu'à n
    ↪ inclus *)
let somme_entiers n =
  let rec boucle somme i =
    if i <= n then
      boucle (somme+i) (i+1)
    else
      somme
  in
  boucle 0 0
```

Ici, la boucle C a été transformée en la fonction `boucle`. Dans la boucle C, on a $\texttt{somme}' = \texttt{somme} + \texttt{i}$ et $\texttt{i}' = \texttt{i} + 1$. La version récursive fonctionne en calculant ces valeurs prime (donc en « simulant une itération »), puis en s'appelant elle-même pour simuler les itérations suivantes.

C'est une transformation classique, que l'on fera souvent lorsque l'on travaille en OCaml.

Remarque. La transformation inverse (récursion -> boucle) est possible, mais plus technique lorsqu'il y a plusieurs appels récursifs (il faut simuler la pile d'appels à la main).

1.3.1 Récursivité terminale (hors-programme)

Si une fonction récursive s'appelle *une seule fois*, et que cet appel est *la toute dernière chose* effectuée, alors il est très simple de transformer la fonction récursive en boucle. C'est une optimisation faite par le compilateur OCaml[7] !

Cela permet de gagner :

- un peu de temps, car passer d'une itération d'une boucle à la suivante est plus rapide que d'ouvrir un appel récursif
- beaucoup d'espace, car il n'y a pas besoin d'utiliser de l'espace pour chacun des appels successifs.

⚠ Écrire une fonction récursive terminale demande parfois de modifier le code de manière peu lisible. Il faut d'abord faire une version lisible et correcte et ensuite seulement une version optimisée.

2 Comment concevoir une fonction récursive

Proposition 3 (Concevoir une solution récursive).
Pour concevoir une solution récursive à un problème sur une instance I :

- On suppose que l'on a une boite noire magique (la récursion) qui peut résoudre toutes les instances du problème sauf I.
- On cherche un lien entre I et une ou plusieurs autres instances du problème. Souvent, cela implique de « reconnaitre le problème dans le problème » (comme pour les triangles) ou de « exprimer $f(n)$ à l'aide de $f(n-1)$ ».
- On cherche les cas où le lien trouvé ne peut pas être utilisé : ce sont les cas de base, et on doit en trouver une solution sans récurrence.
- On vérifie que les instances « diminuent » d'un appel sur l'autre, afin de garantir que la suite d'appel finit par atteindre un cas de base et termine.

7. Et que gcc peut faire si on le lui demande, et que cela n'entre pas en conflit avec d'autres optimisations.

3 Analyse de fonctions récursives

Les méthodes pour analyser la terminaison, la correction et la complexité d'une fonction récursive sont dans les cours en question.

4 Compléments

4.0 Élimination des appels redondants

Lorsque l'on écrit des fonctions récursives, il arrive que certaines instances plus petites soient résolues plusieurs fois. Voici par exemple une fonction (très peu recommandée) pour calculer $\binom{n}{k}$:

exemples.ml

```
24 (** Renvoie k parmi n *)
25 let rec binom n k =
26   if k = 0 || k = n then
27     1
28   else
29     ( binom (n-1) k ) + ( binom (n-1) (k-1) )
```

Si on applique cette méthode et que l'on essaye par exemple de calculer $\binom{5}{2}$, certains appels sont calculés plusieurs fois :

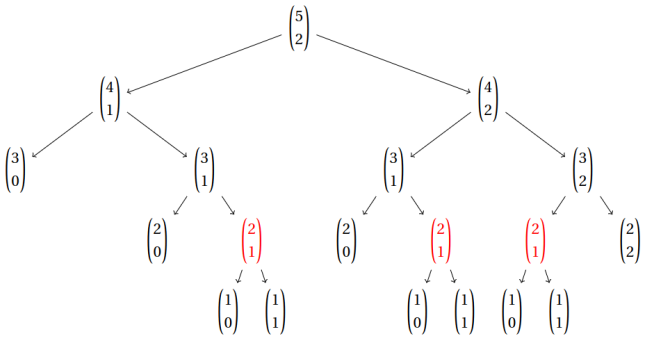

FIGURE 4.7 – Arbre d'appels de `binom 5 2`. Une redondance d'appels est mise en exergue en rouge. (src : J. Larochette)

Ces appels redondants ralentissent le fonctionnement, puisque l'on recalcule plusieurs fois la même chose. Il est plus efficace de mémoriser ces valeurs pour ne pas avoir à les recalculer ensuite : c'est l'objet de la programmation dynamique, que nous verrons au second semestre.

4.1 Avantages et inconvénients de la récursion

Avantages. La récursivité :

- peut produire un code plus lisible et plus simple à comprendre.
- produit un code souvent plus simple à prouver (car les hypothèses de récurrence / invariants sont plus simples).
- permet de résoudre des problèmes très difficiles à résoudre sans !!
- est très adaptée au travail sur les structures de données naturellement récursives (dont les arbres).
- est très adaptée aux problèmes naturellement récursifs.

Inconvénients. La récursivité :

- est plus éloignée du fonctionnement de la machine, et est ralentie par le temps nécessaire à l'ouverture d'un appel de fonction.
- utilise de l'espace mémoire en empilant les appels.
- produit *parfois* un code moins clair qu'un code itératif[8] plus direct.

En résumé, c'est un outil extrêmement puissant. Comme tout outil, il n'est pas absolu et a ses limites d'utilisation. Comme tout bon outil, il rend simple des tâches difficiles. Un grand pouvoir implique de grandes responsabilité !

4.2 Querelle sémantique

Différents termes existent pour parler de la récursivité :

- « récursivité », « fonction récursive » sont les termes usuels en français.
- « récurrence » est également utilisé (c'est de toute façon la même notion qu'en mathématiques).
- « récursion » est une anglicisme du terme anglais « recursion », souvent utilisé.
- « induction », « fonction inductive » sont des anglicismes venant du terme anglais « induction », qui signifie « récurrence ».

À titre personnel, j'utiliserai et mélangerai toutes ces versions. Je connais des auteurs/autrices qui restreignent certains termes[9] : adaptez-vous à ce que vous dit la personne avec qui vous intéragissez.

8. Un code itératif est un code non-récursif.
9. J'ai hésité à le faire aussi, mais je juge qu'à notre modeste niveau, le gain gagné par ces subtilités ne vaut pas l'effort de distinction.