Université de Liège

INFO0947

Compléments de Programmation

Un exercice dont vous êtes le Héros · l'Héroïne.

Récursivité – Calcul d'une Exponentielle

Simon Liénardy Benoit Donnet

30 mars 2020

Préambule

Exercices

Dans ce « TP dont vous êtes le héros », nous vous proposons de suivre pas à pas la résolution de deux exercices. L'un sur un problème concernant seulement un unique fichier et l'autre sur les problèmes sur plusieurs fichiers.

Il est dangereux d'y aller seul [1] !

Partagez vos commentaires, questions, solutions alternatives sur le forum eCampus. N'hésitez jamais !

1. Référence vidéoludique bien connue des Héros.

4.1 Énoncé

Soit a, un entier et n un entier positif ou nul. On se propose de calculer a^n en utilisant la propriété suivante :

2. $a^n = \begin{cases} (a^2)^{\lfloor n/2 \rfloor} & \text{si n>0 est pair,} \\ a \times (a^2)^{\lfloor n/2 \rfloor} & \text{si n est impair,} \\ 1 & \text{si n=0.} \end{cases}$

4.1.1 Méthode de résolution

Nous allons suivre l'approche constructive vue au cours. Pour ce faire nous allons :

1. Formuler le problème récursivement (Sec. 4.2) ;
2. Spécifier le problème complètement (Sec. 4.3) ;
3. Construire le programme complètement (Sec. 4.4) ;
4. Rédiger le programme final (Sec. 4.5).
5. Établir la complexité temporelle du programme final (Sec. 4.6)

4.2 Formulation récursive

Tout d'abord, il convient de formaliser le problème de manière récursive.

Indice : relisez bien l'énoncé avant de progresser dans la suite !

Si vous voyez directement comment procéder, voyez la suite 4.2.2
Si vous êtes un peu perdu, voyez le rappel sur la récursivité 4.2.1

4.2.1 Rappels sur la Récursivité

4.2.1.1 Formulation Récursive

On ne va pas se cacher derrière son petit doigt, c'est la principale étape et la plus difficile. Les étapes suivantes de résolution sont rendues beaucoup plus simples dès que la formulation récursive est connue. Comment procéder ?

En fait, il y a deux choses à fournir pour définir récursivement quelque chose :

1. Un cas de base ;
2. Un cas récursif.

Trouver le **cas de base** n'est pas toujours facile, certes mais ce n'est pas le plus compliqué. En revanche, cerner le **cas récursif** n'est pas une tâche aisée pour qui débute dans la récursivité. Pourquoi ? Parce que c'est complètement contre-intuitif ! Personne ne pense récursivement de manière innée. C'est une *manière de voir le monde*[2] qui demande un peu d'entrainement et de pratique pour être maitrisée. Heureusement, **vous êtes au bon endroit** pour débuter ensemble.

Or donc, toutes les définitions récursives suivent le même schéma :

> Un X, c'est *quelque chose* **combiné** avec un un X *plus simple*.

Il faut détailler :
— Quel est ce *quelque chose* ?
— Ce qu'on entend par **combiné** ?
— Ce que signifie plus simple ?
Si on satisfait à ces exigences, on aura défini X récursivement puisqu'il intervient lui-même dans sa propre définition.

Un exemple mathématique, pour illustrer ce canevas de définition récursive :

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

On veut définir ici la factorielle de n ($n!$). On dit que c'est n (notre *quelque chose*) combiné par la multiplication à une factorielle plus simple : $(n-1)!$

4.2.1.2 Programmatiquement parlant

Par définition, un programme récursif s'appelle lui-même, sur des données *plus simple*. La tâche du programmeur consiste donc à déterminer comment **combiner** le résultat de l'appel récursif sur ces données plus simples avec les paramètres du programme pour obtenir le résultat voulu.

Dans la rédaction d'un programme récursif, on a donc **toujours** accès à un sous-problème particulier : le programme lui-même. Il convient par contre de respecter deux règles (cf. slides Chap. 4, 23 → 26) :

> **Règle 1** Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif ;
>
> **Règle 2** Tout appel récursif doit se faire avec des données plus « proches » de données satisfaisant une condition de terminaison

2. On dit aussi *paradigme*.

Les cas non-récursifs sont appelés **cas de base**.

Les conditions que doivent satisfaire les données dans ces cas de base sont appelées **conditions de terminaison**.

4.2.1.3 Construction récursive d'un programme

Pour construire un programme récursif, il convient de respecter ce schéma. Notez bien qu'ici n'interviendra pas la notion d'invariant tant qu'il n'y a pas de boucle en jeu [3] !

1. Formuler récursivement le Problème ;
2. Spécifier le problème. Le plus souvent, il suffit d'utiliser la formulation récursive préalablement établie ;
3. Construire le programme final en suivant l'approche constructive. La plupart du temps, c'est trivial, il suffit de suivre la formulation récursive.

Et les sous-problème dans tout ça ? Si le problème principal semble trop compliqué, il se peut qu'une découpe en sous-problèmes soit pertinente. Ce sont alors ces sous-problèmes qui seront récursifs et dont le programme pourra être rédigé en suivant les trois points ci-dessus.

De toute façon, chaque programme récursif possède au moins un sous-problème : lui-même ! Par définition de la récursion. Attention que par l'approche constructive, avant d'appeler n'importe quel sous-problème, il convient d'assurer que sa précondition est respectée. Après l'appel, sa postcondition sera respectée par l'approche constructive.

4.2.1.4 Suite de l'exercice

À vous ! Formalisez le problème de manière récursive et passez à la section 4.2.2.

3. Peut-on mêler Invariant et récursion ? Bien sûr : souvenez-vous en lorsqu'on vous parlera de l'algorithme de tri `quicksort`.

4.2.2 Mise en commun de la formulation récursive

En fait l'énoncé lui-même nous donne une formalisation récursive du problème !

$$a^n = \begin{cases} (a^2)^{\lfloor n/2 \rfloor} & \text{si n} > 0 \text{ est pair,} \\ a \times (a^2)^{\lfloor n/2 \rfloor} & \text{si n est impair,} \\ 1 & \text{si n} = 0. \end{cases}$$

D'un côté, on essaie de définir a^n, en fonction d'une version plus simple de l'exponentielle. Ici, on a une exponentielle plus simple parce **l'exposant** a été divisé par 2. La **base**, par contre, a été élevée au carré. C'est dans cela que réside la récursivité ! On combine donc le carré de la base a en l'élevant à un exposant deux fois moins grand. On n'oublie évidemment pas le cas de base ($n = 0$).

Pourquoi mentionner $\lfloor n/2 \rfloor$? C'est la division entière de n par 2.
Si n est pair, on a :

$$\begin{aligned} n &= 2 \times k \\ \lfloor n/2 \rfloor &= k \end{aligned}$$

Par contre, si n est impair, on a :

$$\begin{aligned} n &= 2 \times k + 1 \\ \lfloor n/2 \rfloor &= k \\ 2 \times \lfloor n/2 \rfloor &= 2 \times k \\ 1 + 2 \times \lfloor n/2 \rfloor &= n \end{aligned}$$

Si n est impair, la dernière équation ci-dessous montre bien pourquoi il faut multiplier une fois a dans le second cas récursif. Sans cela, le résultat serait incorrect.

4.2.2.1 Suite de l'exercice

Vous commencez à connaitre le fonctionnement de la résolution : il faut maintenant passer aux spécifications. Voir la Sec. 4.3.

4.3 Spécification

Il faut maintenant spécifier le problème.

Pour un rappel sur la spécification, voyez la section 4.3.1
La correction de la spécification est disponible à la section 4.3.3

4.3.1 Rappel sur les spécifications

Voici un rappel à propos de ce qui est attendu :

La Précondition C'est une condition sur les données. Pour l'établir, il faut se demander quelles doivent être les données du problème. Ici, dans notre exemple, cela parle de tableau. Quelles conditions devons-nous imposer sur ce tableau ? N'hésitez pas à vous référer au cours INFO0946, Chapitre 6, Slides 54 → 64).

La Postcondition C'est une condition sur le résultat du problème, si tant est que la Précondition est remplie (si tel n'est pas le cas, on pratique la *programmation défensive* (voir INFO0946, Chapitre 6, Slides 65 → 70) et on arrête l'exécution du programme.

La signature est souvent nécessaire lors de l'établissement de la Postcondition des fonctions (voir INFO0946, Chapitre 6, Slide 11). En effet, le résultat retourné par l'invocation de la fonction est représenté par le nom de la fonction dans la Postcondition. Le nom de la fonction doit donc être défini avant d'en parler.

4.3.2 Suite de l'exercice

Spécifiez le problème. rendez-vous à la section 4.3.3 pour la correction !

4.3.3 Spécification du problème

Une fois n'est pas coutume, il faut essentiellement faire attention à la Précondition ! Il faut interdire à l'utilisateur de calculer n'importe quoi. Soyons prudents !

Précondition On interdit de calculer l'exponentielle 0^0, qui correspond à un cas d'indétermination. On exprime donc que $n = 0$ implique que la base ne soit pas nulle.

$$Pre : n \geq 0 \wedge (n = 0 \Rightarrow a \neq 0)$$

Postcondition On remarque que le but du programme est de calculer a^n. Il n'y a aucune raison de ne pas réutiliser cette notation fort convenable :

$$Post : \texttt{power} = a^n$$

Alerte : Studentite aigüe !

Si vous n'avez pas une Postcondition aussi simple, c'est que vous avez contracté une *studentite aigüe*, la maladie caractérisée par une inflammation de l'étudiant qui cherche midi à quatorze heures.

Heureusement, ce n'est pas dangereux tant que vous êtes confiné ! Faites une petite pause dans cette résolution. Prenez le temps de vous oxygéner, en faisant un peu d'exercice physique par exemple. Trois répétitions de dix pompes devraient faire l'affaire.

Signature On choisit de représenter les entiers signés avec des `long` (`int` est ok, bien entendu). On impose un n non-signé.

```
1 long power(long a, unsigned int n);
```

Suite

Passez maintenant à la construction du code : Sec. 4.4.

4.4 Construction du programme par l'approche constructive

Il faut maintenant construire le programme en suivant *l'approche constructive* La section suivante vous fournit un petit rappel.

Le corrigé est disponible à la section 4.4.2.

4.4.1 Approche constructive et récursion

Les principes de base de l'approche constructive ne changent évidemment pas si le programme construit est récursif :

- Il faut vérifier que la **Précondition** est respectée en pratiquant la **programmation défensive** ;
- Le programme sera construit autours d'une instruction conditionnelle qui discrimine le·s cas de base des cas récursifs ;
- Chaque **cas de base** doit amener la **Postcondition** ;
- **Le cas récursif** devra contenir (au moins) un **appel récursif** ;
- Pour tous les appels à des sous-problème (y compris l'appel récursif), il faut vérifier que la **Précondition** du module que l'on va invoquer est respectée
- Par le principe de l'approche constrictive, la **Postcondition** des sous-problèmes invoqués est respectée après leurs appels.

Il faut respecter ces quelques règles en écrivant le code du programme. La meilleure façon de procéder consiste à suivre d'assez près la formulation récursive du problème que l'on est en train de résoudre. Le code est souvent une « traduction » de cette fameuse formulation. Il ne faut pas oublier les **assertions intermédiaires**. Habituellement, elles ne sont pas tellement compliquées à fournir.

Quant est-il de `INIT`, `B`, `CORPS` et `FIN` ?

Essayez de suivre ! Ces zones correspondent au code produit lorsqu'on écrit une boucle ! S'il n'y a pas de boucle, il n'y aura pas, par exemple, de gardien de boucle !

Et la fonction de terminaison ?

Là non plus, s'il n'y a pas de boucle, on ne peut pas en fournir une. **Par contre**, on s'assure qu'on a pas oublié le·s **cas de base** et que les différents **appels récursifs** convergent bien vers un des cas de base (la convergence vers un des cas de base est appelée **condition de terminaison** – Chapitre 4, Slides 22 → 27).

Suite de l'exercice

À vous maintenant de construire le programme par l'approche constructive. Un corrigé est disponible à la Sec. 4.4.2.

4.4.2 Approche constructive

4.4.2.1 Programmation défensive

On n'oublie pas de vérifier que la Précondition est remplie en interdisant à a et n d'êtres nuls en même temps.

```
int power(int a, unsigned n){
    assert(a != 0 || n != 0);
    // {Pre}
```

4.4.2.2 Cas de base

On gère le cas de base où $n = 0$. Juste avant, on gère le cas précis où $a = 0$ (on est assuré que n n'est pas nul dans ce cas)

```
// {Pre}
if (!a)
    // a = 0 ⇒ n ∅
    return 0;
    // {Post}
// a ≠ 0
if (n == 0)
    // a ≠ 0 ∧ n = 0
    return 1;
    //n = 0 ⇒ a^n = a^0 = 1 ⇒ {Post}
```

4.4.2.3 Cas récursifs

Il y a deux cas récursifs, selon que n est pair ou impair. On teste donc le reste de la division de n par 2. $\{Pre_{REC}\}$ et $\{Post_{REC}\}$ sont respectivement la Précondition et la Postcondition de l'appel récursif. Les assertions intermédiaires rappellent la discussion faite sur n à la Sec. 4.2.2.

```
else if (n % 2)
    // n mod 2 ∧ a ≠ 0 ⇒ {Pre_REC}
    return a * power(a*a, n/2);
    // {Post_REC} : power = (a^2)^(n/2)  (1)
    // n mod 2 ⇒ n = 2k + 1 ∧ k = n/2  (2)
    // De (1) et (2) :  a × power = a × (a^2)^k = a^(2k+1) = a^n ⇒ {Post}
else
    // ¬(n mod 2) ∧ a ≠ 0 ⇒ {Pre_REC}
    return power(a*a, n/2);
    // {Post_REC} : power = (a^2)^(n/2)  (1)
    // ¬(n mod 2) ⇒ n = 2k ∧ k = n/2  (2)
    // De (1) et (2) : power = (a^2)^(n/2) = (a^2)^k = a^2k = a^n ⇒ {Post}
}
```

Suite de l'exercice

Le programme final est donné à la section suivante : 4.5. Il sera utile pour l'évaluation de la complexité (Sec. 4.6).

4.5 Programme final

```
1  // @Pre : n ≥ 0 ∧ (n = 0 ⇒ a ≠ 0)
2  // @Post: power = a^n
3  int power(int a, unsigned n){
4     assert(a != 0 || n != 0);
5
6     if (!a)
7        return 0;
8     if (n == 0)
9        return 1;
10
11    else if (n % 2)
12       return a * power(a*a, n/2);
13
14    else
15
16       return power(a*a, n/2);
17 }
```

Suite de l'exercice

Comme vous pouvez le constater, tous les numéros de lignes ont été indiqué pour faciliter la discussion au sujet de la complexité. Passez donc à la section 4.6.

4.6 Évaluation de la complexité du programme final

4.6.1 Rappel sur l'évaluation Complexité des modules récursifs

Le but de l'évaluation de la complexité consiste toujours à étudier la quantité de ressources utilisées par un programme. En ce qui concerne les programmes récursifs, on va surtout essayer de compter le nombre d'appels récursifs causés par l'exécution du programme. La plupart du temps, les calculs impliqueront de résoudre une **équation de récurrence** (cfr. Chapitre 4, Slides 66 → 67).

4.6.1.1 Méthode

1. Définir $T(n)$ qui est le temps d'exécution pour un appel de la fonction récursive.
2. n est donc évidemment le paramètre récursif (= une grandeur manipulée dans le code, le plus souvent une variable)
3. Exprimer $T(n)$ dans le cas de base. Le plus souvent, c'est une constante
4. Exprimer $T(n)$ dans le cas récursif. Le plus souvent, $T(\cdot)$ apparait dans cette expression, mais avec un paramètre plus simple, correspondant à l'appel récursif.
5. Résoudre le système d'équation ainsi modélisé (cfr. Chapitre 4, Slides 68 → 70), soit par :

— Élimination de la récurrence de proche ne proche ;
— Identification de la solution et démonstration de celle-ci ;
— Utilisation de solution d'équations connues.

Alerte : Pourquoi tant de n ?

Ce problème touche ceux qui n'ont pas compris que le n dans $T(n)$ est le *paramètre récursif*, qui dépend donc du module envisagé lors de l'évaluation de la complexité. n `DOIT` donc être une grandeur manipulée par le code. Sinon, la complexité obtenue par les calculs n'aura **aucun sens**

Beaucoup trop de rapports de projets ont été affectés ces derniers temps. Soyez bien prudents. Protégez-vous !

4.6.1.2 Élimination de la récurrence de proche en proche

Illustrons cette méthode avec l'exemple de la factorielle :

```
1 unsigned int fact(unsigned int n){
    if (n <= 1)
3     return 1;
    else
5     return n * fact(n-1);
  }
```

En ce qui concerne le cas de base, on va dire qu'il prend a opérations, a étant constant. Pour ce qui est du cas récursif, on va dire qu'il demande b opérations ainsi que le nombre d'opérations nécessaires par l'appel récursif, c'est-à-dire $T(n-1)$. Il vient le système suivant :

$$\begin{array}{rcll} T(n) & = & a & \text{si } n <= 1 \\ & = & b + T(n-1) & \text{sinon} \end{array}$$

L'élimination de proche en proche consiste à réécrire plusieurs fois cette équation en utilisant le cas récursif afin de faire apparaitre un motif reconnaissable.

$$\begin{aligned} T(n) &= b + T(n-1) \\ &= b + b + T(n-2) \\ &= 2b + T(n-2) \\ &= 3b + T(n-3) \\ &= 4b + T(n-4) \\ &= \dots \\ &= kb + T(n-k) \end{aligned}$$

$T(n-k)$ est une formulation tout à fait générale du temps d'exécution du k^{e} appel récursif. On ne connait qu'une valeur particulière de $T(\cdot) : T(1) = 0$. On va donc essayer de faire apparaitre cette valeur particulière. On a :

$$\begin{aligned} n - k &= 1 \\ -k &= 1 - n \\ k &= n - 1 \end{aligned}$$

On insère cette valeur dans $kb + T(n-k)$, il vient :

$$T(n) = (n-1)b + T(1) = (n-1)b + a \in \mathcal{O}(n)$$

Suite de l'exercice

Il faut maintenant calculer la complexité de notre programme. La mise en commun se situe à la Sec. 4.6.2.

4.6.2 Complexité de la fonction `power`

$$T(n) = \begin{cases} a & \text{si n} = 0\,; \\ b + T(\frac{n}{2}) & \text{sinon.} \end{cases}$$

On peut écrire ceci bien qu'il y ait deux cas récursifs. En fait, on considère qu'ils vont impliquer autant d'opérations élémentaires : b. Par contre, le temps pris par l'appel récursif est bien $T(n/2)$. Il y a bien une addition entre b et $T(n/2)$ parce qu'il faut d'abord faire b opérations, **puis**, il faut exécuter 1 appel récursif.

Réécrivons plusieurs fois l'équation :

$$\begin{aligned} T(n) &= b + T(\frac{n}{2}) \\ &= b + b + T(\frac{n}{4}) \\ &= 2 \times b + T(\frac{n}{4}) \\ &= 3 \times b + T(\frac{n}{8}) \\ &= 4 \times b + T(\frac{n}{16}) \\ &\dots \\ &= k \times b + T(\frac{n}{2^k}) \end{aligned}$$

Il est plutôt malaisé de faire apparaitre $T(0)$ tout de suite mais on sait que

$$T(1) = b + T(0) = b + a$$

Essayons de faire apparaitre $T(1)$ à la place de $T(n/2^k)$:

$$\begin{aligned} \frac{n}{2^k} &= 1 \\ n &= 2^k \\ \log n &= k \end{aligned}$$

Il vient alors :

$$T(n) = k \times b + T(\frac{n}{2^k}) = b \times \log n + b + a \in \mathcal{O}(\log n)$$

Nous avons donc une complexité d'ordre logarithmique.