Université de Liège

INFO0947

Compléments de Programmation

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

Récursivité – Tableaux

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, tous deux concernant la récursion. Un premier exercice porte sur l'exponentiation, l'autre sur la manipulation d'un tableau.

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

Spécifiez et construisez une fonction récursive renvoyant la position d'un entier donné dans un tableau `t` d'entiers donné (-1 si le nombre ne s'y trouve pas).

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).

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 Sec. 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

Ici, il y a clairement (au moins) deux écoles pour décrire récursivement un tableau !

4.2.2.1 Première méthode

On peut définir un tableau récursivement comme ceci :

> Un tableau de taille N, c'est *une case* **suivie** d'un tableau de taille N-1.

Ce qui correspond à ce schéma :

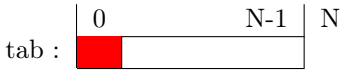

FIGURE 1 – Un tableau composé d'une cellule suivi d'un tableau plus petit

4.2.2.2 Deuxième méthode

On peut définir un tableau récursivement comme ceci :

> Un tableau de taille N, c'est un tableau de taille N-1 **suivie** d'une emphune case.

Ce qui correspond à ce schéma :

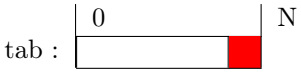

FIGURE 2 – Un tableau composé d'un tableau plus petit suivi d'une cellule

Les deux définitions sont évidemment équivalentes mais ne mèneront pas au même programme ! Choisissez votre camp une fois pour toute !

Alerte : Difficulté !

La première méthode est clairement plus complexe et le code qui en découle, non optimisé, sera moins efficace que celui obtenu par la seconde méthode. À vos risques et périls !

4.2.2.3 Suite de l'exercice

Pour la formulation récursive suivant la première méthode voir la section 4.2.2.4
Pour la formulation récursive suivant la seconde méthode voir la section 4.2.2.5

4.2.2.4 Formulation suivant la 1re méthode

$$Cherche\_rec(T, N, X) = \begin{cases} -1 & \text{si } N = 0 \quad (1) \\ 0 & \text{si } N \neq 0 \wedge T[0] = X \quad (2) \\ -1 & \text{si } N > 0 \wedge Cherche\_rec(T+1, N-1, X) = -1 \quad (3) \\ 1 + Cherche\_rec(T+1, N-1, X) & \text{sinon} \quad (4) \end{cases}$$

Dans cette formulation, on envisage le tableau comme étant une case suivie d'un tableau plus petit. On teste donc la première case du tableau `T[0]`. On a les 4 cas suivants :

1. Si le tableau est vide, on renvoie -1 ;
2. Si X est dans la case de tête, on renvoie son indice ;
3. Si l'appel récursif renvoie -1, on propage ce résultat dans le code appelant ;
4. On exécute d'abord l'appel récursif. Le tableau situé derrière la case de tête commence à l'adresse $T+1$ et à une taille $N-1$. X ne change évidemment pas. Puisque le résultat de l'appel récursif renvoie un indice décalé d'une case, on rattrape le décalage en ajoutant 1 à cet indice.

Suite de l'exercice

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

4.2.2.5 Formulation suivant la 2e méthode

$$Cherche\_rec(T, N, X) = \begin{cases} -1 & \text{si } N = 0 \quad (1) \\ N - 1 & \text{si } N \neq 0 \wedge T[N-1] = X \quad (2) \\ Cherche\_rec(T, N-1, X) & \text{sinon } (i.e.\ T[N-1] \neq X) \quad (3) \end{cases}$$

Dans cette formulation, on envisage le tableau comme étant un tableau plus petit suivi d'une case. On teste donc la première case du tableau `T[0]`. On a les 3 cas suivants :

1. Si le tableau est vide, on renvoie -1 ;
2. Si la dernière case du tableau contient l'élément recherché, on renvoie ce dernier indice $N - 1$;
3. Sinon, on lance la recherche dans le tableau plus petit. Son adresse n'a pas changé (pas de modification du pointeur !) mais sa taille, bien : il fait maintenant $N - 1$ cases. X ne change évidemment pas !

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. Bien qu'il y ait deux méthodes de résolution, la spécification devrait être identique !

Pour un rappel sur la spécification, voyez la section 4.3.1

La correction de la spécification est disponible à la section

4.3.1 Rappel sur les spécifications

On s'attend à fournir trois informations :

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.

Suite de l'exercice

En avant ! Spécifiez le problème et rendez-vous à la Sec. 4.3.2.

4.3.2 Spécification du problème

Précondition On impose seulement que la taille du tableau soit positive ou nulle (une taille nulle correspond au cas de base de notre problème).

$$Pre : N \geq 0 \wedge T[0..N-1]init$$

Postcondition On réutilise la notation précédemment définie, évidemment ! La formulation peut être explicitée de deux manière différente mais cela ne change rien à la Postcondition.

$$Post : \texttt{cherche\_rec} = Cherche\_rec(T, N, X)$$

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.

Si vous avez **déjà** contracté la studentite aigüe précédemment, vous risquez d'atteindre le stade chronique ! Vous vous engagez donc dans un traitement prophylactique [a] de fond : trois répétitions de 20 pompes.

a. Se dit de quelque chose qui prévient une maladie

Signature On passe l'adresse du tableau `T`, sa taille `N`, ainsi que l'élément que l'on recherche, `X`. On renvoie un entier signé puisque -1 doit pouvoir être représenté.

```
1 int cherche_rec(int * T, unsigned int N, int X);
```

4.4 Construction du programe

Il faut maintenant construire le programme en suivant l'approche constructive La Sec. 4.4.1 vous fournit un petit rappel.

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

Construisez maintenant votre programme par l'approche constructive.
En suivant la première méthode, rendez-vous à la section 4.4.2
En suivant la seconde méthode, rendez-vous à la section 4.4.3

4.4.2 Approche constructive – Première Méthode

4.4.2.1 Programmation défensive

Il suffit de tester l'adresse du tableau.

```
int cherche_rec(int *T, unsigned int N, int X){
    assert(T);
    // {Pre}
```

4.4.2.2 Cas de base

On teste les deux cas de base : le tableau vide ($N = 0$) et la première case du tableau ($T[0]$).

```
if (N == 0)
    // N = 0
    return -1;
    // Cherche_rec(T, N, X) = -1 ⇒ {Post}

// N ≠ 0
if (T[0] == X)
    // T[0] = X ∧ N ≠ 0
    // Cherche_rec(T, N, X) = 0
    return 0;
    // {Post}
```

4.4.2.3 Cas récursifs

Il faut d'abord récupérer le résultat de l'appel récursif (attention aux paramètres) dans une variable intermédiaire. Soit X n'a pas été trouvé et il faut propager cette information en retournant 1, soit on l'a trouvé dans le tableau plus petit et il faut corriger l'indice obtenu qui correspond au tableau plus grand. $\{Pre_{REC}\}$ et $\{Post_{REC}\}$ sont respectivement la Précondition et la Postcondition de l'appel récursif.

```
// N > 0 ⇒ {Pre_REC}
int resultat = cherche_rec(T + 1, N + 1, X);
// {Post_REC}resultat = Cherche_rec(T + 1, N + 1, X)
if (resultat == -1)
    // Cherche_rec(T + 1, N + 1, X) = -1
    return -1;
    // {Post}
else
    // resultat = Cherche_rec(T + 1, N + 1, X)
    return 1 + resultat;
    // Cherche_rec(T, N, X) = 1 + Cherche_rec(T + 1, N + 1, X)
    // {Post}
```

4.4.3 Approche constructive – Seconde Méthode

4.4.3.1 Programmation défensive

Il suffit de tester l'adresse du tableau.

```
int cherche_rec(int *T, unsigned int N, int X){
   assert(T);
   // {Pre}
```

4.4.3.2 Cas de base

Il y a deux cas de base : le tableau vide ($N = 0$) et le test de la dernière case du tableau

```
if (N == 0)
   // n == 0
    return -1;
    // cherche_rec(T, 0, X) = −1 ⇒ {Post}

// n > 0
if (T[N-1] == X)
   // T[N − 1] = X ⇒ cherche_rec(T, N, X) = N − 1
   return N-1;
    // {Post}
```

4.4.3.3 Cas récursifs

Si on a pas trouvé X dans le dernier élément, on lance la recherche sur le tableau composé d'une case de moins. $\{Pre_{REC}\}$ et $\{Post_{REC}\}$ sont respectivement la Précondition et la Postcondition de l'appel récursif.

```
else
// N > 0 ∧ T[N − 1]not = x
// T[0 . . . N − 1] init ∧ n > 0 ⇒ T[0 . . . N − 2] init ≡ {Pre_REC}
   return cherche_rec(T, N - 1, X);
    // {Post_REC}( ≡ si cherche_rec = Cherche_rec(T, N, X)
}
```

4.5 Programmes finaux

4.5.1 Par la première méthode

```
int cherche_rec(int *T, unsigned int N, int X){
   assert(T);
   // Pre

   if (N == 0)
      // N = 0
      return -1;
      // Cherche_rec(T, N, X) = -1 ⇒ Post

   // N ≠ 0
   if (T[0] == X)
      // T[0] = X ∧ N ≠ 0
      // Cherche_rec(T, N, X) = 0
      return 0;
      // Post

   // N > 0 ⇒ Pre_REC
   int resultat = cherche_rec(T + 1, N + 1, X);
   // Post_REC resultat = Cherche_rec(T + 1, N + 1, X)
   if (resultat == -1)
      // Cherche_rec(T + 1, N + 1, X) = -1
      return -1;
      // Post
   else
      // resultat = Cherche_rec(T + 1, N + 1, X)
      return 1 + resultat;
      // Cherche_rec(T, N, X) = 1 + Cherche_rec(T + 1, N + 1, X)
      // Post
}
```

4.5.2 Par la seconde méthode

```
int cherche_rec(int *T, unsigned int N, int X){
   assert(T);
   // {Pre}

   if (N == 0)
      // n == 0
      return -1;
      // cherche_rec(T, 0, X) = -1 ⇒ {Post}

   // n > 0
   if (T[N-1] == X)
      // T[N - 1] = X ⇒ cherche_rec(T, N, X) = N - 1
      return N-1;
      // {Post}
   else
   // N > 0 ∧ T[N - 1]not = x
   // T[0 . . . N - 1] init ∧ n > 0 ⇒ T[0 . . . N - 2] init ≡ Pre_REC
      return cherche_rec(T, N - 1, X);
      // Post_REC( ≡ si cherche_rec = Cherche_rec(T, N, X)
}
```