Université de Liège

INFO0946

Introduction à la Programmation

GameCode : Un exercice dont vous êtes le Héros · l'Héroïne

Récursivité – Tableaux

Benoit Donnet Simon Liénardy

24 février 2021

Préambule

Exercices

Dans ce GameCode, nous vous proposons de suivre pas à pas la résolution d'un exercice portant sur la récursivité.

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 Rappels sur la Récursivité

4.1.1 Généralités

Un algorithme de résolution d'un problème P sur une donnée a est dit *récursif* si parmi les opérations utilisées pour le résoudre, on trouve une résolution du même problème P sur une donnée b.
On parle d'*appel récursif* toute étape de l'algorithme résolvant le même problème sur une autre donnée.

En pratique, b sera toujours "plus petit" que a. Par exemple dans le cadre des dérivées (en analyse mathématique), pour dériver $(u+v)$ (i.e., $(u+v)'$), il faut additionner la dérivée de u (u') à la dérivée de v (v'). Soit $(u'+v')$. Dans cet exemple, la donnée "plus petite" est le calcul de la dérivée sur les fonctions u et v.

De manière générale, en informatique, un module (fonction ou procédure) est récursive si son exécution peut conduire à sa propre invocation (i.e., le corps du module contient une invocation au module lui-même).

4.1.2 Règles de Conception

Règle 1 : **Distinguer plusieurs cas**. Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif. Cela signifie donc que le squelette général d'un algorithme récursif sera basé sur une instruction conditionnelle. Soit

```
1    int fonction_recursive(int x){
2      if(B)
3        //cas sans appel récursif
4        return ...;
5      else
6        //cas avec appel(s) récursif(s)
7        return fonction_recursive(...);
8    }
9
```

Le cas non récursif (i.e., la clause "Alors") dans l'extrait de code ci-dessus s'appelle le *cas de base*. La condition qui doit être satisfaite pour exécuter le cas de base (i.e., `B` dans l'extrait de code ci-dessus) s'appelle la *condition de terminaison*.

Règle 2 : **Appel récursif "plus petit"**. Tout appel récursif doit se faire avec des données plus "proches" de données satisfaisant une condition de terminaison. Cette règle est clairement une application relative à la terminaison d'un programme, i.e., il ne peut y avoir une infinité d'appels récursifs. Le programme doit se terminer un jour. Ce qui donne :

```
1  int fonction_recursive(int x){
2    if(B)
3      //cas sans appel récursif
4      return ...;
5    else
6      //cas avec appel(s) récursif(s) où x' est "plus petit" que x
7      return fonction_recursive(x');
8  }
9
```

La signification de "plus proche" ou "plus petit" dépend clairement du contexte. Par exemple, si `x` est un tableau, alors `x'` est un tableau plus petit. Ou encore, si `x` est un naturel, alors `x'` $<$ `x`.

A titre d'exemple, voici une version récursive du calcul de la factorielle :

```
1 int factorielle(int n){
2   if(n==0)
3     return 1;
4   else
5     return n * factorielle(n-1);
6 }
```

Dans cet exemple, l'expression `n==0` est la condition de terminaison. L'instruction `return 1;` est le cas de base. Enfin, l'instruction `return n * factorielle(n-1);` est l'appel récursif puisqu'on dispose d'une invocation de la fonction `factorielle()` sur un paramètre effectif plus petit que le paramètre formel (`n-1` $<$ `n`). Dit autrement, chaque appel récursif tend vers le cas de base.

4.1.3 Exécution Récursive

L'exécution d'une fonction/procédure récursive se fait, typiquement, en deux étapes :

1. *Descente récursive*. Il s'agit de l'étape qui consiste à empiler les appels récursifs, jusqu'à atteindre la condition de terminaison (et donc exécuter le cas de base). Le coût de la descente récursive est donc la création, en cascade, de contextes sur la pile (1 contexte par appel récursif). Eventuellement, la descente récursive peut impliquer un calcul.
2. *Remontée récursive*. Il s'agit de l'étape de nettoyage de la pile : après avoir atteint le cas de base, les différents contextes sont supprimés un par un jusqu'à revenir au code appelant initial. Eventuellement, la remontée récursive peut nécessiter, avant chaque suppression de contexte, un calcul.

Dans l'exemple de la factorielle, la descente récursive a pour seul but d'accumuler sur la pile toutes les valeurs intermédiaires entre n et 0. La remontée récursive permettra de faire les calculs cumulatifs (i.e., les produits intermédiaires) sur base des valeurs empilées lors de la descente.

4.1.4 Types de Récursivité

Il existe 5 types de récursivité :

1. Récursivité *simple*. Un algorithme récursif est simple ou linéaire si chaque cas se résout en au plus un appel récursif. Par exemple :

```
int factorielle(int n){
  if(n==0)
    return 1;
  else
    return n * factorielle(n-1);
}

```

2. Récursivité *Multiple*. Un algorithme récursif est multiple si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs. Par exemple :

```
void hanoi(int n, tour *d, tour *a, tour *i){
  if(n==1)
    deplacer_disque(d, a);
  else{
    hanoi(n-1, d, i, a);
    deplacer_disque(d, a);
    hanoi(n-1, i, a, d);
  }
}

```

3. Récursivité *Croisée*. Deux algorithmes sont mutuellement récursifs si l'un fait appel à l'autre et l'autre fait appel à l'un. La récursivité croisée nécessite une déclaration forward (ou l'utilisation d'un header). Par exemple :

```
int I(int n);

int P(int n){
  if(n==0)
    return 1;
  else
```

```
    rerturn I(n-1);
}

int I(int n){
  if(n==0)
    return 0;
  else
    return P(n-1);
}

```

4. Récursivité *Terminale*. Un algorithme est récursif terminal si l'appel récursif est la dernière instruction de la fonction. La valeur retournée est directement obtenue par un appel récursif. C'est donc lors de la descente récursive que les calculs sont exécutés. La remontée récursive ne sert qu'à nettoyer la pile. Par exemple :

```
int f(int n, int a){
  if(n==0)
    return a;
  else
    return f(n-1, n*a);
}

```

5. Récursivité *Imbriquée*. Un algorithme est récursif imbriqué si l'appel récursif contient lui aussi un appel récursif. Un algorithme récursif imbriqué est, très souvent, extrêmement inefficace. Par exemple :

```
int ackermann(int m, int n){
  if(m==0)
    return n+1;
  else{
    if(n==0)
      return ackermann(m-1, 1);
    else
      return ackermann(m-1, ackermann(m, n-1));
  }
}

```

4.2 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).

4.2.1 Méthode de Résolution

On va procéder à la résolution de ce problème en suivant pas à pas l'approche constructive :

1. Formuler de manière récursive le problème (Sec. 4.3) ;
2. Spécifier le problème complètement (Sec. 4.4) ;
3. Construire le programme complètement (Sec. 4.5) ;
4. Rédiger le programme final (Sec. 4.6) ;

4.3 Formulation Récursive

La première étape dans la résolution du problème nécessite sa formalisation sous forme récursive.
Si vous voyez de quoi on parle, rendez-vous à la Section 4.3.3
Si vous ne savez comment introduire une formalisation récursive, voyez la Section 4.3.1
Si vous ne voyez pas comment faire, reportez-vous à l'indice 4.3.2

4.3.1 Formulation Récursive

Il y a deux informations 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.

Toutes les définitions récursives suivent le même schéma :

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

Prenons un exemple pour illustrer ce canevas de définition récursive :

> La factorielle de n ($n!$) vaut 1 si n est nul. Sinon, c'est n **multiplié** par la factorielle de $n-1$.

Le cas de base est assez évident : quand $n = 0$, la factorielle de n est 1. Pour le cas récursif, on dit que c'est n (notre *quelque chose*) combiné par la **multiplication** à une factorielle plus simple : $(n-1)!$

Mathématiquement, cela donne :

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

Suite de l'Exercice

À vous ! Formalisez le problème et passez à la Sec. 4.3.3.

Si vous séchez, reportez-vous à l'indice à la Sec. 4.3.2

2. On dit aussi *paradigme*.

4.3.2 Indice

Prenez le temps de bien relire l'énoncé. Il y a deux façons de définir récursivement le problème, ce qui mènera à deux programmes différents (l'un étant plus efficace que l'autre).

Suite de l'Exercice

À vous ! Formulez récursivement le problème et passez à la Sec. 4.3.3.

4.3.3 Mise en Commun de la Formalisation du Problème

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

1. Première méthode :

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

2. Deuxième méthode :

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

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 !

Quelque soit la définition choisie, le format général sera le suivant :

$$Cherche\_Rec(T, N, X) = \dots$$

Notre notation s'appelle $Cherche\_Rec$ et prend trois informations en entrée : T, le tableau, N, sa taille et X, la valeur à chercher.

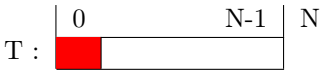
- Formulation récursive suivant la première méthode ;
- Formulation récursive suivant la deuxième méthode.

4.3.3.1 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 :



On peut formellement exprimer cela de la façon suivante :

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

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 pouvez maintenant passer à la suite de l'exercice :

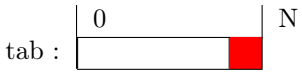
- Formulation récursive suivant la deuxième méthode ;
- Écriture de la spécification pour résoudre le problème.

4.3.3.2 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 *une case*.

Ce qui correspond à ce schéma :



On peut formellement exprimer cela de la façon suivante :

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

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 pouvez maintenant passer à la suite de l'exercice :

- Formulation récursive suivant la première méthode ;
- Écriture de la spécification pour résoudre le problème.

4.4 Spécifications

Il s'agit, maintenant, de proposer une interface pour la fonction.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.4.3
Si vous ne savez pas ce qu'est une spécification, allez à la Section 4.4.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.4.2

4.4.1 Rappels sur les Spécifications

4.4.1.1 Généralités

Une spécification se définit en deux temps :

La PréCondition implémente les suppositions. Elle caractérise donc les conditions initiales du module, i.e., les propriétés que doivent respecter les valeurs en entrée (i.e., paramètres effectifs) du module. Elle se définit donc sur les paramètres formels (qui seront initialisés avec les paramètres effectifs). La PréCondition doit être satisfaite avant l'exécution du module.

La PostCondition implémente les certifications. Elle caractérise les conditions finales du résultat du module. Dit autrement, la PostCondition décrit le résultat du module sans dire comment il a été obtenu. La PostCondition sera satisfaite après l'invocation.

A l'instar de la définition d'un problème, un module doit toujours avoir une PostCondition (un module qui ne fait rien ne sert à rien) mais il est possible que le module n'ait pas de PréCondition (e.g., le module ne prend pas de paramètres formels).

La spécification d'un module se met, en commentaires, avec le prototype du module. L'ensemble (i.e., spécification + prototype) forme l'*interface* du module.

Contrairement au cours INFO0946, les spécifications d'un module seront, dorénavant, exprimées de manière formelle (i.e., à l'aide de prédicats et de notations).

4.4.1.2 Construction d'une Interface

Pour construire l'interface d'un module, il est souhaitable de commencer par faire un dessin. Le dessin, qui naturellement sera lié à Invariant Graphique, doit représenter des situations particulières du problème à résoudre. La situation initiale (i.e., avant la première instruction de la Zone 1/Init) doit aider pour trouver le prototype du module, ainsi que la PréCondition. La situation finale (i.e., après exécution de la Zone 3/End) doit vous aider à trouver la PostCondition.

En s'appuyant sur les dessins, il suffit ensuite de répondre à trois questions pour construire l'interface d'un module :

Question 1 : Quels sont les objets dont j'ai besoin pour atteindre mon objectif ? Répondre à cette question permet de construire le prototype du module (éventuel type de retour, identificateur du module, paramètres formels).

Question 2 : Quel est l'objectif du module ? Répondre à cette question permet d'obtenir la PostCondition, représentée à l'aide d'un prédicat.

Question 3 : Quelles sont les contraintes sur les paramètres ? Répondre à cette question permet d'obtenir la PréCondition, représentée à l'aide d'un prédicat.

L'avantage de faire un dessin est, bien entendu, de dresser un pont avec l'Invariant Graphique, mais aussi de faciliter la production d'un prédicat (il "suffit" de traduire le dessin en un prédicat).

4.4.1.3 Exemple

A titre d'exemple, essayons de spécifier une fonction qui permet de calculer le produit de tous les entiers entre des bornes fournies, i.e., `a` et `b` (avec $b > a$).

Le problème peut s'illustrer comme indiqué à la Fig. 1. On dessine une droite (représentant les nombres qu'on va manipuler) avec les bornes de l'intervalle d'intérêt. La flèche bleue indique ce qu'on doit calculer.

Question 1
Pour atteindre l'objectif, nous avons simplement besoin des bornes de l'intervalle. Sur la Fig. 1, on peut clairement voir que le calcul du produit s'étale entre `a` et `b`. Ce sont les seuls éléments "extérieurs" dont on a besoin pour résoudre le problème. `a` et `b` seront donc les paramètres formels du module.

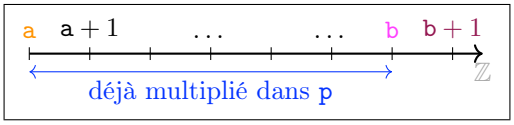


FIGURE 1 – Exemple de représentation graphique du problème à résoudre.

La Fig. 1 illustre aussi le fait qu'on manipule des entiers ($\mathbb{Z}$). a et b seront donc de type `int`.

Enfin, on peut adéquatement nommer le module `produit()`, ce qui permet de naturellement décrire ce qu'il fait.

Tout ceci nous permet de dériver le prototype du module :

```
1 produit(int a, int b);
```

Question 2

L'objectif du module est de calculer (et donc retourner) le produit de tous les entiers dans l'intervalle $[a, b]$. Soit :

$$a \times (a+1) \times \cdots \times (b-1) \times b$$

On peut représenter ce produit de manière plus condensée en utilisant le quantificateur numérique du produit : $\prod$. Soit :

$$\prod_{i=a}^{b} i$$

Puisque a et b sont de type `int`, le produit résultat sera aussi de type `int`. Le module `produit()` est donc une fonction retournant un `int`. On représente cela de la façon suivante :

$$produit = \prod_{i=a}^{b} i$$

Enfin, `a` et `b` ne sont pas modifiés par le module.

A ce stade, on obtient donc l'interface incomplète suivante :

```
1 /*
2  * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
3  */
4 int produit(int a, int b);
```

Question 3

Enfin, nous pouvons déterminer si il existe des conditions sur les paramètres. C'est bien le cas ici. L'énoncé du problème et la Fig. 1 indiquent clairement que b est strictement plus grand que a. On peut le représenter de la façon suivante :

$$b > a$$

Interface
Au final, l'interface du module est la suivante :

```
/*
 * PréCondition : b > a
 * PostCondition : produit = ∏_{i=a}^{b} i ∧ a = a_0 ∧ b = b_0
 */
int produit(int a, int b);
```

Suite de l'Exercice

À vous ! Proposez une interface pour la fonction et passez à la Sec. 4.4.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 4.4.2

4.4.2 Indice

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

Ne soyez pas ambigu dans la PréCondition, ni dans la PostCondition. Au contraire, soyez le plus précis possible (rappelez-vous, un prédicat est une formulation mathématique et, dès lors, ne peut souffrir d'une quelconque ambiguïté). On n'oubliera pas de s'appuyer sur la notation récursive introduite.

Suite de l'Exercice

À vous ! Spécifiez la fonction et passez à la Sec. 4.4.3.

4.4.3 Mise en Commun des Spécifications

Commençons par la PréCondition. On impose seulement que la taille du tableau soit positive ou nulle (une taille nulle correspond au cas de base de notre problème). Ce qui nous donne :

$$\text{PréCondition} \equiv N \geq 0 \wedge T[0...N-1]\ init$$

L'objectif de la fonction est de déterminer la présence d'une valeur dans le tableau et retourner son éventuelle position. Nous allons donc, logiquement, l'appeler `cherche_rec()`. Elle prendra trois paramètres en entrée : `T`, le tableau (qui contient des valeurs entières), `N`, la taille du tableau ($\in \mathbb{N}$) et `X`, la valeur à chercher.

Passons à la 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. Les paramètres formels ne sont pas modifiés dans le code.

$$\text{PostCondition} \equiv cherche\_rec = Cherche\_Rec(T, N, X) \wedge T = T_0 \wedge N = N_0 \wedge X = X_0$$

On voit bien ici, grâce à la formulation récursive, que la fonction retournera une valeur entière (-1 si `X` ne se trouve pas dans `T`).

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

Au final, l'interface de la fonction est :

```
1 /*
2  * PréCondition : N ≥ 0 ∧ T[0...N − 1] init
3  * PostCondition : cherche_rec = Cherche_Rec(T, N, X) ∧ T = T0 ∧ N = N0 ∧ X = X0
4  */
5 int cherche_rec(int * T, unsigned int N, int X);
```

Suite de l'exercice

Vous pouvez maintenant passer à la construction récursive de votre fonction.

4.5 Construction du Code

Une fois la spécification proposée, on peut construire le code en adoptant l'approche constructive et on précisant, à chaque étape, les assertions intermédiaires.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.5.3
Si vous ne savez pas comment fonctionne l'approche constructive appliquée à la récursivité, allez la Section 4.5.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.5.2

4.5.1 Approche Constructive et Récursivité

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.

Qu'en est-il de Init, B, Iter et End ? 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 (i.e., chaque appel récursif tend vers la condition de terminaison).

Suite de l'Exercice

À vous ! Construisez le code pour la fonction et passez à la Sec. 4.5.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 4.5.2

4.5.2 Indice

Il faut repartir de la structure générale d'un algorithme récursif en prenant soin de déterminer les assertions intermédiaires à chaque étape.

On n'oubliera pas de bien vérifier que la PréCondition est validée avant l'appel récursif.

On procédera en trois étapes : (*i*) programmation défensive, (*ii*) cas de base, (*iii*) cas récursif(s).

Attention, n'oubliez pas que vous devez baser votre implémentation sur l'une des deux formulations récursives.

Suite de l'Exercice

À vous ! Construisez le code pour la fonction et passez à la Sec. 4.5.3.

4.5.3 Mise en Commun de l'Approche Constructive

Les deux formulations récursives conduisent à deux implémentations différentes.

- Construction du code en suivant la première méthode ;
- Construction du code en suivant la deuxième méthode.

4.5.3.1 Construction et Première Méthode

La structure générale d'une fonction/procédure récursive s'appuie sur une structure conditionnelle. On va remplir donc cette structure en trois étapes : (*i*) programmation défensive, (*ii*) cas de base, (*iii*) cas récursif(s).

On va aussi s'appuyer sur la première formulation récursive du problème

Programmation Défensive

La PréCondition est la suivante :

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

Il suffit donc de tester l'adresse du tableau (`N` est déclaré comme `unsigned int`).

```
int cherche_rec(int *T, unsigned int N, int X){
  assert(T);
  // {PréCondition ≡ N ≥ 0 ∧ T[0..N − 1] init}
}
```

Cas de Base

La formulation récursive nous indique deux cas de base : le tableau vide ($N = 0$) et la première case du tableau par rapport à X ($??T[0] == X??$).

```
// {PréCondition ≡ N ≥ 0 ∧ T[0...N − 1] init}
if(N == 0)
  // {N = 0 ∧ N = N0 ∧ T = T0 ∧ X = X0}
  return -1;
  // {Cherche_Rec(T, N, X) = −1 ∧ N = N0 ∧ T = T0 ∧ X = X0}
  // {⇒ PostCondition}
// {N > 0}
if(T[0] == X)
  // {T[0] = X ∧ N ≠ 0 ∧ T = T0 ∧ X = X0 ∧ N = N0}
  return 0;
  // {cherche_rec(T, N, X) = 0 ∧ T = T0 ∧ X = X0 ∧ N = N0}
  // {⇒ PostCondition}
```

Cas Récursif(s)

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. $\{\text{PréCondition}_{REC}\}$ et $\{\text{PostCondition}_{REC}\}$ sont respectivement la PréCondition et la PostCondition de l'appel récursif.

```
// {N > 0 ∧ T[0...N − 1]init}
// {⇒ PréCondition_REC}
int resultat = cherche_rec(T + 1, N + 1, X);
// {PostCondition_REC ≡ resultat = Cherche_Rec(T + 1, N + 1, X) ∧ T = T0 ∧ X = X0 ∧ N = N0}
if (resultat == -1)
    // {resultat = Cherche_Rec(T + 1, N + 1, X) = −1 ∧ T = T0 ∧ X = X0 ∧ N = N0}
    return -1;
    // {cherche_rec() = −1 ∧ T = T0 ∧ X = X0 ∧ N = N0}
    // {⇒ PostCondition}
else
    // {resultat = Cherche_Rec(T + 1, N + 1, X) ∧ T = T0 ∧ X = X0 ∧ N = N0}
```

```
12      return 1 + resultat;
13      // {Cherche_Rec(T, N, X) = 1 + Cherche_Rec(T + 1, N + 1, X) ∧ T = T0 ∧ X = X0 ∧ N = N0}
14      // {⇒ PostCondition}
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- Construction du code en suivant la deuxième méthode ;
- Rédaction du programme final.

4.5.3.2 Construction et Deuxième Méthode

La structure générale d'une fonction/procédure récursive s'appuie sur une structure conditionnelle. On va remplir donc cette structure en trois étapes : (i) programmation défensive, (ii) cas de base, (iii) cas récursif(s).

On va aussi s'appuyer sur la deuxième formulation récursive du problème

Programmation Défensive

La PréCondition est la suivante :

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

Il suffit donc de tester l'adresse du tableau (`N` est déclaré comme `unsigned int`).

```
int cherche_rec(int *T, unsigned int N, int X){
  assert(T);
  // {PréCondition ≡ N ≥ 0 ∧ T[0..N − 1] init}
}
```

Cas de Base

La formulation récursive nous indique deux cas de base : le tableau vide ($N = 0$) et le test de la dernière case du tableau ($??T[N-1] == X??$).

```
if(N == 0)
  // {N = 0 ∧ T = T0 ∧ X = X0 ∧ N = N0}
  return -1;
  // {Cherche_Rec(T, N, X) = −1 ∧ N = N0 ∧ T = T0 ∧ X = X0}
  // {⇒ PostCondition}

// {N > 0}
if(T[N-1] == X)
  // {T[N − 1] = X ∧ N = N0 ∧ T = T0 ∧ X = X0}
  return N-1;
  // {cherche_rec = N − 1 ∧ N = N0 ∧ T = T0 ∧ X = X0}
  // {⇒ PostCondition}
```

Cas Récursif(s)

Si on a pas trouvé X dans le dernier élément, on lance la recherche sur le tableau composé d'une case de moins. $\{\text{PréCondition}_{REC}\}$ et $\{\text{PostCondition}_{REC}\}$ 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 ≡ PréCondition_REC}
  return cherche_rec(T, N - 1, X);
  // {PostCondition_REC ≡ cherche_rec = Cherche_Rec(T, N − 1, X) ∧ T = T0 ∧ X = X0 ∧ N = N0 ∧ T[N − 1] ≠ x}
  // {cherche_rec = Cherche_Rec(T, N, X) ∧ T = T0 ∧ X = X0 ∧ N = N0}
  // {⇒ PostCondition}
}
```

Suite de l'exercice

Vous pouvez maintenant passer à la suite de l'exercice :

- Construction du code en suivant la première méthode ;
- Rédaction du programme final.

4.6 Code Complet

- Code complet en suivant la première méthode ;
- Code complet en suivant la deuxième méthode.

4.6.1 Première Méthode

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

  if(N == 0)
    return -1;

  if(T[0] == X)
    return 0;

  int resultat = cherche_rec(T + 1, N + 1, X);

  if (resultat == -1)
    return -1;
  else
    return 1 + resultat;
}//fin cherche_rec()
```

4.6.2 Deuxième Méthode

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

  if(N == 0)
    return -1;

  if(T[N-1] == X)
    return N-1;
  else
    return cherche_rec(T, N - 1, X);
}//fin cherche_rec()
```