Université de Liège

INFO0946

Introduction à la Programmation

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

## Récursivité – Calcul d'une Exponentielle

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

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

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

### 4.2.1 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) ;
5. Établir la complexité théorique du programme final (Sec. 4.7).

## 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

Pour définir récursivement un problème, il faut d'abord identifier le cas de base. Normalement, une "simple" relecture de l'énoncé devrait vous amener la solution.

Ensuite, pour le cas récursif, il faut identifier, dans l'énoncé, ce qui est combiné avec quelque chose de plus petit pour résoudre le problème !

#### 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

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.

#### Suite de l'exercice

Vous pouvez maintenant proposer une 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.

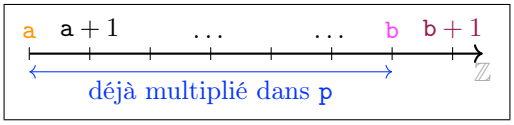


FIGURE 1 – Exemple de représentation graphique du problème à résoudre.

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

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 (e.g., $0^0$, ce qui correspond à un cas d'indétermination). Soyons prudents !

Pour le reste, il suffit d'agir comme d'habitude, à savoir répondre aux trois questions. La spécification de la fonction est, ici, assez simple. On peut raisonnablement (pour une fois) se passer d'un schéma.

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 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. On dispose donc, en entrée, de deux valeurs. On obtient, alors, la PréCondition suivante :

$$\text{PréCondition} \equiv n \geq 0 \wedge (n = 0 \Rightarrow a \neq 0)$$

L'objectif de la fonction est de calculer une puissance. Nous allons donc, logiquement, l'appeler `power()`. Choisissons pour `a` le type `unsigned long` (`unsigned int` fonctionne aussi). Par contre, on impose que `n` soit `unsigned int`.

Passons à la 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 :

$$\text{PostCondition} \equiv power = a^n \wedge a = a_0 \wedge n = n_0$$

Attention, le calcul de $a^n$ peut générer des nombres très grands. Il est donc préférable de déclarer que la fonction `power` retourne une valeur de type `unsigned long`.

**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.

Au final, l'interface de la fonction est :

```
1 /*
2  * PréCondition : n ≥ 0 ∧ (n = 0 ⇒ a ≠ 0
3  * PostCondition : power = a^n ∧ a = a_0 ∧ n = n_0
4  */
5 unsigned long power(unsigned long a, unsigned int n);
```

#### 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).

#### 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

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

#### 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.

```
unsigned long power(unsigned long a, unsigned int n){
  assert(a != 0 || n != 0);
  // {PréCondition ≡ n ≥ 0 ∧ (n = 0 ⇒ a ≠ 0)}
}
```

#### 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)

```
// {PréCondition ≡ n ≥ 0 ∧ (n = 0 ⇒ a ≠ 0)}
if (!a)
  // {a = 0 ⇒ n ≠ 0}
  return 0;
  // {a = a0 ∧ n = no ∧ power = 0 = 0^n}
  // {⇒ PostCondition}
// {a ≠ 0}
if (n == 0)
  // {a ≠ 0 ∧ n = 0}
  return 1;
  // {a = a0 ∧ n = n0 ∧ power = 1 = a^0 = a^n}
  // {⇒ PostCondition}
```

#### Cas Récursif(s)

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. {PréCondition$_{REC}$} et {PostCondition$_{REC}$} sont respectivement la PréCondition et la PostCondition de l'appel récursif. Les assertions intermédiaires rappellent la discussion faite sur $n$.

```
else
  if(n % 2)
    // {n mod 2 ∧ a ≠ 0}
    // {⇒ PréCondition_REC}
    return a * power(a*a, n/2);
    // {PostCondition_REC ≡ power = a^2)^{n/2} ∧ n mod 2 ⇒ n = 2k + 1 ∧ k = n/2 ∧ a = a0 ∧ n = n0}
    // {a × power = a × (a^2)^k = a^{2k+1} = a^n ∧ a = a0 ∧ n = n0}
    // {⇒ PostCondition}
  else
    // {¬(n mod 2) ∧ a ≠ 0}
    // {⇒ PréCondition_REC}
    return power(a*a, n/2);
    // {PostCondition_REC ≡ power = (a^2)^{n/2} ∧ ¬(n mod 2) ⇒ n = 2k ∧ k = n/2 ∧ a = a0 ∧ n = n0}
    // {power = a^2)^{n/2} = (a^2)^k = a^{2k} = a^n ∧ a = a0 ∧ n = n0}
    // {⇒ PostCondition}
}
```

#### Suite de l'exercice

Vous pouvez maintenant rédiger le programme final.

## 4.6 Code Complet

```
int power(int a, unsigned n){
  assert(a != 0 || n != 0);
  if(!a)
    return 0;
  if(n == 0)
    return 1;
  else
    if (n % 2)
      return a * power(a*a, n/2);
    else
      return power(a*a, n/2);
}//fin power()
```

### Suite de l'exercice

Vous pouvez maintenant évaluer la complexité.

## 4.7 Complexité Théorique

Une fois le code rédiger, on peut évaluer l'efficacité du code en étudiant la complexité théorique du programme.

Si vous voyez de quoi on parle, rendez-vous à la Section 4.7.3
Si vous ne savez pas comment évaluer la complexité des modules récursifs 4.7.1
Enfin, si vous ne voyez pas quoi faire, reportez-vous à l'indice 4.7.2

### 4.7.1 Rappels sur la Complexité des Modules Récursifs

#### 4.7.1.1 Généralités

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*.

Etant donné une suite de nombres $(t(1), t(2), \ldots, t(n), \ldots)$, une équation reliant le $n^{\text{me}}$ terme à ses prédécesseurs est appelée équation de récurrence.
La résolution d'une équation de récurrence consiste à trouver une expression du $n^{\text{me}}$ en fonction du paramètre $n$.

#### 4.7.1.2 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, soit par :
   - Élimination de la récurrence de proche en 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**

Illustrons l'élimination de la récurrence de proche en proche avec l'exemple de la factorielle :

```
1 unsigned int fact(unsigned int n){
2   if(n <= 1)                          } —— a
3     return 1;                  } —— b
4   else
5     return n * fact(n-1);      } —— T(n − 1)
6 }
```

Soit $T(n)$, le coût d'un appel à `fact(n)`.
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 :

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

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.

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

$T(n-k)$ 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 $k \times b + T(n-k)$, il vient :

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

**Suite de l'Exercice**

À vous ! Évaluez la complexité de la fonction et passez à la Sec. 4.7.3.
Si vous séchez, reportez-vous à l'indice à la Sec. 4.7.2

### 4.7.2 Indice

Pour évaluer la complexité théorique d'une fonction/procédure, il faut faire l'inventaire des instructions exécutées par celle-ci.

Le plus simple est de découper la fonction/procédure en différents blocs. Dans le cadre de la récursivité, les différents blocs sont constitués autour de l'instruction conditionnelle : cas de base vs. cas récursif.

Il faut ensuite exprimer le tout sous la forme d'une équation de récurrence et la résoudre.

#### Suite de l'Exercice

À vous ! Évaluez la complexité de la fonction et passez à la Sec. 4.7.3.

### 4.7.3 Mise en Commun de la Complexité

Le corps de la fonction, avec découpe en régions, est le suivant :

```
1 if(!a)
2   return 0;
3 if(n == 0)
4   return 1;
5 else
6   if (n % 2)
7     return a * power(a*a, n/2);
8   else
9     return power(a*a, n/2);
```

— $a$ (lignes 3–4) ; — $b$ (lignes 5–6) ; — $T(n/2)$ (ligne 7) ; — $T(n/2)$ (lignes 8–9)

On peut faire une telle découpe, bien qu'il y ait deux cas de base. En fait, on considère qu'ils vont impliquer autant d'opérations élémentaires : $b$. Par contre, le temps pris par l'appel récursif est bien $T(n/2)$.

On obtient alors le système suivant :

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

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.

Résolvons l'équation de récurrence par application de la méthode de proche en proche :

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

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.