IUT De Villetaneuse
Dép[t] informatique

Cours d'algorithmique
2$^{\text{éme}}$ Année

Cours 8

Programmation récursive

---

1. Qu'est-ce que la programmation récursive

Définition : la programmation récursive est une technique de programmation qui remplace les instructions de boucle (while, for, etc.) par des appels de fonction.

(Ne pas confondre avec la notion de récursivité en mathématiques)

1.1. Premier exemple

Le mécanisme le plus simple pour faire boucler une fonction : elle se rappelle elle-même.
Exemple de définition de fonction qui se rappelle elle même :

```
void boucle() {
    boucle();
}
```

On peut en profiter pour faire quelque chose :

```
void boucle() {
    printf ("Je tourne \n") ;
    boucle() ;
}
```

C'est ce qu'on appelle une récursivité simple. Il faut encore ajouter un mécanisme de test d'arrêt. Ex. écrire 100 fois "Je tourne" : on a besoin d'un compteur. On choisit ici de le passer d'un appel de fonction à l'autre comme un paramètre.

```
void boucle(int i) {
    if (i < 100) {
           printf ("Je tourne \n") ;
           boucle(i+1) ;
    }
    //sinon on ne relance pas => fin du programme
}
```

et il faut appeler **boucle(0)**.

1.2. Apprendre à programmer récursivement avec des variables

Calculer la somme des n premiers nombres avec une boucle while :

```
int somme(int n) {
    int s = 0 ;
    int i = 1 ;
    while (i <= n) {
        s = s+i ; i=i+1 ;}
    return s ;
}
```

par une procédure récursive , méthode **très naïve** :

```
int n = 100 ;
int s = 0 ;
void sommeRec(int i) {
      if (i <= n) {
            s = s+i ;
            sommeRec(i+1) ;
      }
      //si i = n, fin du programme
}
et on lance sommeRec(0) ;
```

C'est extrêmement maladroit parce que : 1. on ne contrôle pas la valeur de s au début 2. on ne contrôle pas la valeur terminale N. Pour éviter cela, il faut accéder à s et n sans qu'elles soient en variables globales. Une solution pour construire une procédure récursive sérieuse est d'utiliser pour passer s et n les paramètres et la valeur de retour de la fonction

```
int sommeRec(int i, int s, int n) {
      if (i <= n) {
            return sommeRec(i+1, s+i, n) ;
      }
      else
            return (s) ; //sommeRec(n,s,n), on a fini
}
```

et on lance sommeRec(0, 0 , 100) ;

On aurait pu éviter de passer à la fois i et n, en comptant à l'envers de n à 0 :

```
int sommeRec(int s, int n) {
      if (n > 0) {
            return sommeRec(s + n, n - 1 );
      }
      else
            return (s) ; //Pour sommeRec(0, s), le calcul est immédiat
}
```

On lance x = sommeRec(0, 100).

Et on pouvait même éviter de passer s, en gérant plus efficacement la valeur de retour :

```
int sommeRec(int n) {
      if (n > 0) {
            return sommeRec(n - 1 ) + n;
      }
      else
            return (0) ; //Pour sommeRec(0) ,  le calcul est immédiat
}
```

On lance x = sommeRec(100).

Enfin, on peut s'apercevoir qu'il est plus astucieux de programmer une fonction plus générale : sommeRec(début, fin), c'est "faire la somme des nombres de *début* jusqu'à *fin* ". La programmation récursive, c'est : on appelle la même fonction avec un nombre de moins dans la liste, puis on ajoute ce nombre au résultat. D'où deux versions :

```
int sommeRec(int deb, int fin) {
      if (fin >= deb) {
            return sommeRec(deb, fin - 1 ) + fin;
      }
      else
            return (0) ; //Pour sommeRec(x,x), le calcul est immédiat
}
```

ou

```
int sommeRec(int deb, int fin) {
      if (fin >= deb) {
            return deb + sommeRec(deb + 1, fin );
      }
      else
            return (0) ; //Pour sommeRec(x,x), le calcul est immédiat
}
```

Danns les deux cas, on lance sommeRec(0, 100), sommeRec(10, 50), etc.

2. Différents types de récursivité

2.1. Récursivité simple

Définition : une fonction simplement récursive, c'est une fonction qui s'appelle elle-même une seule fois, comme c'était le cas pour sommeRec() ci dessus.

Le résultat dépend de l'ordre dans lequel on fait les opérations dans le corps de la fonction. Par exemple, on a un tableau de mots que l'on veut afficher par une procédure récursive, on peut faire l'appel récursif avant ou après l'affichage :

```
public class StringRec {
   String tabmots [] ;

   StringRec(String mots[]){
         tabmots = mots ;
   }

   void afficheRecAp(int i) {
         if (i < tabmots.length) {
               System.out.print(tabmots[i] + " ") ;
               afficheRecAp(i+1) ;
         }
   }

   void afficheRecAv(int i) {
         if (i < tabmots.length) {
               afficheRecAv(i+1) ;
               System.out.print(tabmots[i] + " ") ;
         }
   }
}
```

Définition : un appel récursif dans lequel la fonction n'exécute aucune instruction après l'appel est un appel récursif **terminal**.

Ex : afficheRecAp utilise un appel récursif terminal, alors que afficheRecAv utilise un appel non terminal.

2.2. Quelques explications

afficheRecAp et afficheRecAv ne produisent pas le même résultat. Avec tabmots initialisé à {"Dernier", "cours", "algo", "avancee"}, on obtient pour afficheRecAp : "Dernier cours algo avancee", pour afficheRecAv : "avancee algo cours Dernier". afficheRecAp() écrit donc le tableau à l'endroit, alors que afficheRecAv() l'écrit à l'envers.

- Dans les deux cas, on a les appels (X = Ap ou Av) : afficheRecX(0) → afficheRecX(1) → afficheRecX(2) → afficheRecX(3).
- afficheRecAp(i) écrit le mot i, puis lance afficheRecAp(i+1) qui écrit le mot i+1 après le mot i.
- Au contraire, afficheRecAv(i+1) se termine (après avoir écrit le mot i+1) avant que afficheRecAv(i) n'écrive le mot i.
- Dans les deux cas, la fonction appelante se termine après la fonction appelée. Dans un appel terminal, elle se termine immédiatement après, sans rien faire de plus.

Remarque : ça ressemble beaucoup aux parcours d'arbres en profondeur, préfixés ou postfixés. On va voir pourquoi un peu plus loin.

2.3. Récursivité multiple

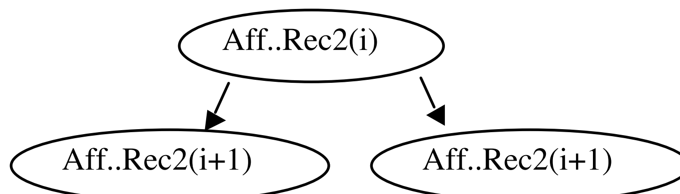
Une fonction peut exécuter plusieurs appels récursifs – typiquement deux, parfois plus. Par exemple :

```
void afficheRec2(int i) {
      if (i < tabmots.length) {
            afficheRec2(i+1) ;
            System.out.print(tabmots[i] + " ") ;
            afficheRec2(i+1) ;
      }
}
```

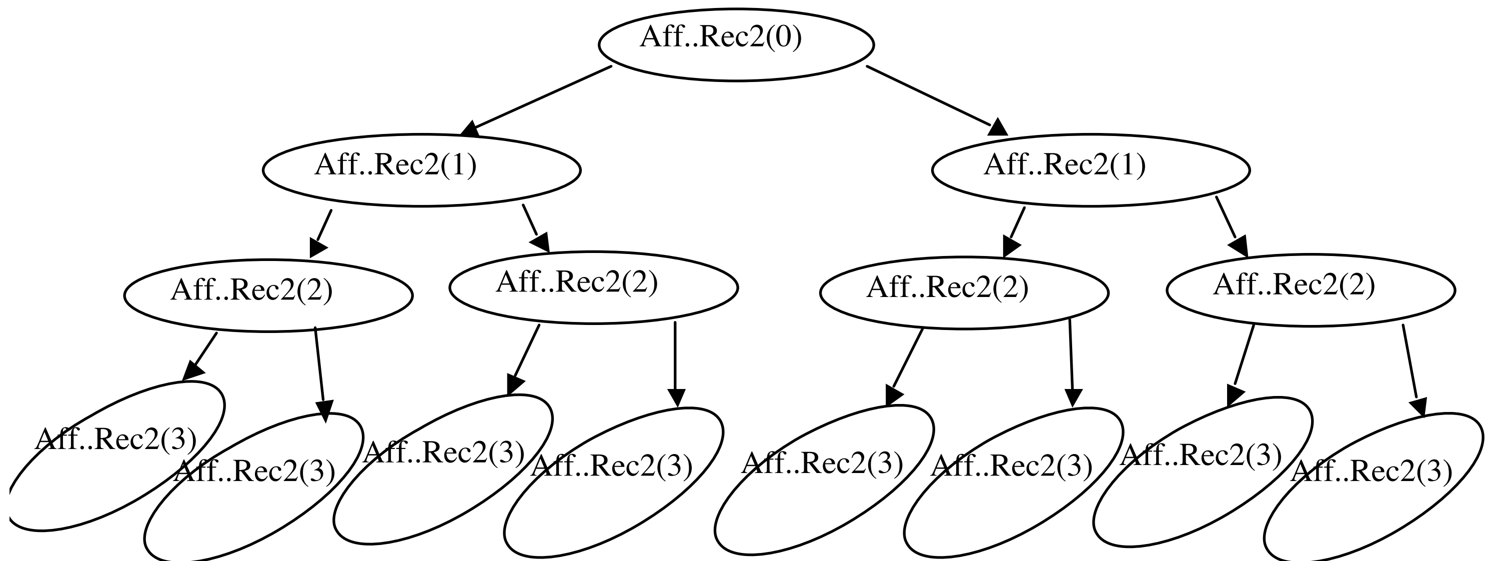
Avec tabmots initialisé à {"Dernier", "cours", "algo", "avancee"}, on obtient pour afficheRec2(0) :

```
avancee algo avancee cours avancee algo avancee Dernier
avancee algo avancee cours avancee algo avancee
```

Que s'est-il passé ? Chaque fonction afficheRec2(i) appelle deux fois afficheRec2(i+1). Si l'on dessine les appels (le premier appel à gauche), on a :



Ce qui, quand on regarde l'ensemble, constitue un arbre.



Il faut analyser un peu plus les appels récursifs pour comprendre l'affichage. Pour l'instant, notez qu'on a bien une fois "dernier", 2 fois "cours", 4 fois "algo" et 8 fois "avancee"

A retenir : dans n'importe quel programme, les appels de fonction forment un arbre (à l'exécution). En C, la racine de l'arbre est ………………. Chaque fonction appelle zéro, une, deux, .., n fonctions (ses fonctions filles), qui à leur tour appellent leurs fonctions filles, etc. Quand l'exécution d'une fonction est finie, on revient à sa fonction mère, puis celle-ci va appeler la prochaine de ses filles. L'exécution réalise donc **un parcours en profondeur d'abord**.

2.4. Appel récursif, arbre et pile

Toute fonction "connaît" un certain nombre de variables et leurs valeurs. On appelle cet ensemble de couples (variable,valeur) le **contexte** de la fonction. Quand on appelle une fonction, ses arguments sont les modifications du contexte existant pour constituer le contexte de la fonction. Ils lui sont passés sur la pile.
Quand la fonction se termine, elle dépile les arguments, et l'on retourne à la fonction appelante qui retrouve son contexte.
Quand on regarde l'arbre des appels ci dessus, le sous arbre sous le fils précédent est entièrement exécuté avant qu'on passe au fils suivant. Autrement dit, comme dans le cas standard, **l'exécution d'un appel récursif est un parcours d'arbre en profondeur d'abord**.

Nous avons appris que pour un parcours en profondeur, il faut une pile. Ici, **on a utilisé implicitement la pile système** (la pile des arguments de fonction) pour revenir au père (à la fonction appelante).

Si l'on condidère afficheRec2(), elle affiche entre les deux appels récursifs – i.e. entre ses fils. On retrouve donc une **énumération infixée**. Sachant que afficheRec2(0) affiche *dernier*, afficheRec2(1) affiche *cours*, …, on retrouve le texte produit par l'énumération infixée.

Note : on peut transformer une procédure récursive terminale en procédure itérative **sans pile**, alors que ce n'est pas possible pour une procédure récursive non terminale.

3. Quelques exemples

3.1. Récursivité simple : parcours de liste

Il faut avoir implanté la liste par une définition récursive (une liste contient une liste en variable membre).

```
public class ListeRec {
Object valeurTete ;
ListeRec reste ;

    String toString(){
        return(valeurTete.toString() + reste.toString()) ;
    }
    String toReverseString(){
        return(reste.toString() + valeurTete.toString()) ;
    }
}
```

3.2. Récursivité simple : recherche dichotomique

On stocke des paires (index, objet) dans un tableau. Le tableau est rangé par index (des chaînes en ordre alphabétique). L'utilisateur fournit un index et on doit lui renvoyer l'objet associé.

```
public class DichoMap {
MapObject table[] ;

    MapObject getObject(String index) {
            return (getObject(index,0,table.length)) ;
    }

    MapObject getObject(String index, int deb, int fin) {
            if(deb > fin) return(null) ;
            int milieu = (deb + fin)/2 ;
            MapObject mo = table[milieu] ;
            int order = mo.getIndex().compareTo(index) ;
            if(order == 0)
                  return (mo) ;
            if (order < 0)
                  return (getObject(index, milieu+1, fin)) ;
            return(getObject(index, deb, milieu-1)) ;
    }
}
```

Notez bien que c'est une récursivité simple (un seul des deux appels est exécuté) et terminale (quel que soit l'appel exécuté, c'est la dernière instruction de la fonction – return n'est pas une instruction)

3.3. Récursivité double

C'est typiquement une situation de parcours d'arbre binaire. Par ex. pour calculer la valeur d'une expression qu'on a transformée en arbre (qu'on pourrait ajouter dans une classe Expression vue en TD).

```
public class arbreCalcul{
String symbol ; // +, - *, / ou un nombre
arbreCalcul filsg, filsd ;

    int getValRec() {
        if (this.estFeuille())
                return(Integer.parseInt(symbol)) ;
        int val1, val2, res ;
        val1 = filsg.getValRec() ;
        val2 = filsd.getValRec() ;
        switch(symbol.charAt(0)) {
          case '+' : res=val1+val2 ;
          case '-' : res=val1-val2 ;
          // etc
        }
    }
}
```

Une récursivité double n'est jamais terminale (le premier appel ne peut pas l'être).

4. Bilan sur la récursivité

4.1. A quoi ça sert

- Dans certains cas, on peut écrire un code très court et bien lisible en utilisant une procédure récursive. C'est commode pour **prototyper** un programme (faire un exemplaire d'essai, de mise au point, sur lequel on ne veut pas passer trop de temps) : on a une pile gratuitement, sans avoir besoin de l'écrire. **Cela n'a d'intérêt que pour les opérations qui utilisent une pile, c'est à dire en gros les parcours d'arbres en profondeur d'abord**.

- La procédure récursive n'est **jamais plus rapide** que la procédure itérative correspondante : fondamentalement, le temps de calcul est surtout déterminé par l'algorithme, et tout algorithme peut s'écrire en itératif ou en récursif. Par contre, la pile système passe en général plus de valeurs que la pile qu'on aurait implémentée (l'adresse de retour, mais aussi les arguments supplémentaires). Elle prend donc un peu plus de temps, et surtout nettement plus de place. Il est très rare qu'on utilise une procédure récursive dans la version définitive d'un programme professionnel.

4.2. Les Warnings

- La programmation récursive, ça ne peut pas faire n'importe quoi. Par exemple, ça ne peut pas simplifier le codage d'un parcours en largeur : on est obligé de réimplémenter la file dans les arguments de fonction, et la pile de l'appel récursif est en plus.

- La plus grosse difficulté, c'est qu'il faut bien connaître les calculs de complexité, parce que le temps de calcul d'une procédure récursive naïve peut être catastrophique. Exemple classique : les nombres de Fibboncci, définis par

$$Fib(0) = Fib(1) = 1, Fib(i) = Fib(i-1) + Fib(i–2) \text{ si } i >2.$$

```
int fibo(int n){
    if(n<2) return(1) ;
    int i=1, f0=1, f1=1, tmp ;
    while(i<n) {
        tmp = f0 + f1 ; f0 = f1 ;
        f1 = tmp ; i++ ;
    }
    return(f1) ;
}
```

On est en O(n)

```
int fiboRec(int n) {
    if(n<2) return(1) ;
    return (fiboRec(n-1) +
    fiboRec(n-2)) ;
}
```

On a gagné 4 lignes, mais on est en $O(2^n)$.
Vous pouvez tester la différence.

On peut aussi écrire une fonction fibonacci récursive en O(n) (encore une fois, tout ce qu'on peut faire en itératif peut être fait aussi en récursif). La difficulté est la même qu'en itératif – question de goût, de style et de langage de programmation.