Récursivité

Par
Roland LANNUZEL, Marketing Technique 4D S.A.
Note technique 4D-200203-08-FR
Version 1

Date 1 Mars 2002

Résumé

La récursivité expliquée pas à pas et niveau par niveau.
Bien utilisée, la récursivité peut nous aider à résoudre simplement des problèmes complexes.

4D Notes techniques

Qu'est-ce que la récursivité ?

En matière de programmation, on peut répondre brièvement qu'il y a récursivité lorsqu'une méthode s'appelle elle-même. Une autre forme de récursivité, un peu plus complexe, et moins « visible » est la récursivité circulaire qui se produit lorsqu'une méthode « A » appelle une méthode « B » qui elle même appelle la méthode « A ». Le nombre de méthode n'est d'ailleurs pas forcément limité à deux… : A->B->C…->A est aussi un cas de programmation récursive, mais qui hélas est parfois involontaire, et dans ce cas, souvent difficile à détecter. De toutes les façons, ce troisième cas est à proscrire pour des raisons évidentes de difficulté à suivre le code, à le débuguer et à le maintenir.

Un cas d'école : La factorielle

L'exemple le plus simple est celui de la fonction « factorielle » qui peut être écrite de façon récursive et ce, même si dans ce cas précis, ce n'est ni la façon la plus rapide ni la plus efficace.

Afin de se rafraîchir la mémoire rappelons que la factorielle(n) - qui se note n! -est égal à 1*2*3*4*…*n ou bien encore n*(n-1)*(n-2)*(n-3)… *1.
De plus factorielle(0) = 1

Partant de là, on peut dire que n! = n * (n-1)!
Puis, n*(n-1)*(n-2)!, etc, jusqu'à ce que (n-x) soit égal à 0.

Une méthode « factorielle » pourrait donc être écrite de la façon ci-dessous :

```
C_ENTIER  LONG($1;$0)

Si ($1<=0)
  $0:=1
Sinon
  $0:=$1*Factorielle ($1-1) `appel récursif
Fin de si
```

Cela fonctionne correctement car les variables locales reçues en paramètres($1, {$2, etc.}) ainsi que la variable renvoyée ($0) sont créées à chaque appel de méthode et sont par conséquent différentes des variables elles-mêmes créées dans la méthode appelante.
Cela est bien entendu vrai également pour les variables locales qui pourraient être déclarées dans ces méthodes.
Il est évident que ce n'est pas la façon la plus simple d'écrire une telle fonction, nous l'avons dit plus haut. Une boucle ferait aussi bien l'affaire mais cela reste un exemple simple de récursivité.

Quand une méthode, appelle une autre méthode...

Lorsqu'une méthode "A" appelle une méthode "B", cette dernière s'exécute et rend la main à la méthode "A" (avec ou sans paramètres entrants ou sortants).

Exemple :

A		Niveau 1 (Methode en cours)
>>	B	Niveau 2 (A appelle B)
A	<<	Niveau 1 (B se termine et rend la main)

Ce principe est immuable et doit être respecté également pour une méthode récursive. Une méthode "A" qui appelle une méthode "A" doit *récupérer* la main, de même qu'une méthode "A" appelée par une méthode "A" doit *rendre* la main. À chaque appel le "niveau" de recursivité augmente, à chaque retour le niveau diminue.

Exemple :

A				Niveau 1 (1er appel)
>>	A'			Niveau 2 (A appelle A)
	>>	A"		Niveau 3 (A appelle A)
		>>	A"'	Niveau 4 (A appelle A)
		A"	<<	Niveau 3 (A se termine et rend la main)
		>>	A"'	Niveau 4 (A appelle A)
		A"	<<	Niveau 3 (A se termine et rend la main)
	A'	<<		Niveau 2 (A se termine et rend la main)
A	<<			Niveau 1 (A se termine et rend la main)

Ci-dessus, les méthodes notées A, A', A" et A"' sont une seule et même méthode : A.

La récursivité ne va pas sans inconvénients. En effet, 4D, comme n'importe quel langage devra gérer des piles et des variables locales qui occuperont de la place en mémoire. Dans certains cas, si le programmeur n'a pas pris la peine de contrôler le niveau des appels, il y a un risque de plantage du programme. (ex : " La pile est pleine… "). Signalons que la "pile" est une zone mémoire ou est stocké (empilé) le "point de retour" d'une méthode une fois celle-ci terminée.

Pour reprendre l'exemple ci-dessus :
- Les variables locales de la méthode A" sont différentes de celles de la méthode A'
- La méthode A" "ignore" les variables de la méthode A'.
- A fortiori, A' ignore les variables de A" car ces dernières n'existent pas en mémoire avant l'appel de la méthode.

Au vu de cet exemple, on comprend qu'il vaut mieux effectuer un contrôle précis de la mémoire et des niveaux lorsque le nombre d'appels récursifs risque de devenir important.

Les exemples que nous allons aborder maintenant ne mettent pas en avant ces inconvénients ni les moyens de

les contourner ou de les éviter, mais offrent des exemples de solutions à des problèmes qui, sans la possibilité de programmation récursive seraient certainement plus ardus à résoudre.

Exemple 1 : Recherche de fichiers sur disques

Il est évident que pour chercher un fichier sur un disque, il faut ouvrir un volume et regarder son contenu, qui comme chacun sait, est constitué de dossiers et de fichiers. Nous en obtiendrons la liste grâce aux commandes LISTE DES DOCUMENTS et LISTE DES DOSSIERS.

Une fois ces listes constituées, nous devrons :
1°) chercher la chaîne souhaitée dans chacun des éléments,
2°) à partir de la nouvelle liste de dossiers effectuer, dossier par dossier, une nouvelle recherche.

Le « niveau» de la recherche récursive dépendra du nombre de dossiers imbriqués les uns dans les autres. Dans le cas d'un disque dur, ce nombre est vraisemblablement raisonnable car il ne s'agit pas de la quantité *totale* de dossiers d'un disque dur, mais de la profondeur maximum de leurs *imbrications*.
(Ex : MonVolume/MesDocuments/MesImages/MesDessins/MaMaison/MaCuisine…)

La méthode ci-dessous vous permettra, en plus de trouver vos fichiers (le but n'est pas de concurrencer les outils de recherche fournis par les systèmes d'exploitation), de connaître le nombre maximum d'imbrications de dossiers dans votre machine.

Ce qui nous donne la méthode ci-dessous :

```
`   *** Méthode FindOnVolumes ***
Si (Nombre de parametres=0)

 TABLEAU TEXTE(◊_Path;1000)
  ◊ID:=0
 TABLEAU TEXTE(_Volumes;0)
 LISTE DES VOLUMES(_Volumes)
  $n:=Taille tableau(_Volumes)

  _Volumes:=1

  $RefWin:=Creer fenetre formulaire([RESULTS];"Search";Dialogue modal )
 DIALOGUE([RESULTS];"Search")

 PROPRIETES PLATE FORME($plateforme;$systeme;$Machine)
 Si ($Plateforme=Windows )
    ◊Separator:="/"
 Sinon
    ◊Separator:=":"
 Fin de si
  ◊Level:=0
  ◊LevelMax:=0

 Si (ok=1) & (◊String#"")

  TOUT SELECTIONNER([RESULTS])
  SUPPRIMER SELECTION([RESULTS])

   FindOnVolumes (_Volumes{_Volumes})    `Appel récursif
```

```
    `on enregistre ce qui n'a pas encore été stocké
    `(les résultats ont été stockés 1000 par 1000, on traite
    `les éléments du tableau non stockés)

   REDUIRE  SELECTION([RESULTS];0)
   SUPPRIMER  LIGNES(◊_Path;◊ID+1;1000)  `on supprime ce qui dépasse :-)
   TABLEAU  VERS  SELECTION(◊_Path;[RESULTS]Value)

   ALERTE("Niveau maximum de hiérarchie : "+Chaine(◊LevelMax))

    `et voilà, c'est fini.
   FERMER  FENETRE

   TOUT  SELECTIONNER([RESULTS])
   VISUALISER  SELECTION([RESULTS])

  Fin de si

Sinon

  C_TEXTE($1)
  TABLEAU  TEXTE($_Files;0)
  TABLEAU  TEXTE($_Folders;0)

    `-----------------------------------------
  LISTE DES DOSSIERS($1;$_Folders)
  $n:=Taille tableau($_Folders)
  Boucle ($i;1;$n)
    `Affichage du message de "progression"
  POSITION  MESSAGE(1;1)
    $NewPath:=$1+◊Separator+$_Folders{$i}
  MESSAGE("-> "+$NewPath+(" "*30))

    `recherche de la  chaîne dans le nom du dossier
   $pos:=Position(◊String;$_Folders{$i})
   Si ($Pos>0)
      ◊ID:=◊ID+1
       ◊_Path{◊ID}:=$1+◊Separator+$_Folders{$i}
    Si (◊ID>=1000)
     REDUIRE  SELECTION([RESULTS];0)
     TABLEAU  VERS  SELECTION(◊_Path;[RESULTS]Value)
      ◊ID:=0
    Fin de si
   Fin de si

    `CALCUL DU NOMBRE MAXI DE DOSSIER IMBRIQUÉS
    `on s'apprête à rappeler la méthode, on incrémente le niveau
    ◊Level:=◊Level+1
   Si (◊Level>◊LevelMax)
      ◊LevelMax:=◊Level
   Fin de si

   FindOnVolumes ($NewPath)     `*** Appel récursif ***

    `Une fois la méthode terminée, on décrémente le niveau
    ◊Level:=◊Level-1
```

```
Fin de boucle

  `----------------------------------------
LISTE DES DOCUMENTS($1;$_Files)
$n:=Taille tableau($_Files)
Boucle ($i;1;$n)

   `recherche de la chaîne dans le nom du fichier
  $pos:=Position(◊String;$_Files{$i})
  Si ($Pos>0)
     ◊ID:=◊ID+1
      ◊_Path{◊ID}:=$1+◊Separator+$_Files{$i}

    Si (◊ID>=1000)
     REDUIRE SELECTION([RESULTS];0)
     TABLEAU VERS SELECTION(◊_Path;[RESULTS]Value)
      ◊ID:=0
    Fin de si

  Fin de si
Fin de boucle


Fin de si
```

Exemple 2 : Comment faire des anagrammes ?

Rappelons qu'une anagramme est une combinaison de lettres effectuée à partir d'une série de lettres quelconques.
Par exemple, NICHE est l'anagramme de CHIEN, de même que « ABCD » est l'anagramme de « BADC »

Dans un premier temps, afin de programmer le plus simplement possible et de nous écarter le moins possible du sujet nous ignorerons les lettres doubles. Les exceptions seront traitées dans un second exemple optimisé, inspiré du premier.

Prenons par exemple la chaîne de caractères « MAISON»

De façon presque instinctive, à partir de cette chaîne, on cherchera toutes les anagrammes commençant par « M », puis celles commençant par « A », puis celles commençant par « I », etc. jusqu'à la lettre « N ».

Imaginons que nous sommes arrivés au « I ». Il nous reste donc les lettres « M, A, S, O et N »

Les anagrammes commençant par « I » sont donc constituées de la lettre « I » elle-même, suivie de toutes les anagrammes possibles à partir des lettres restantes.

Pour trouver les anagrammes de « MASON » on va donc être tenté –à nouveau- de commencer par trouver celles qui commencent par la lettre « M », puis pat la lettre « A », puis par la lettre « S », etc jusqu'à arriver au « N ».

Poursuivons : imaginons que nous en sommes au « A » et, par conséquent que nous cherchons les

anagrammes commençant par « IA » (Le « I » du début, suivi du « A » en cours.) Il nous reste donc « MSON »

Nous devons donc trouver les anagrammes de « MSON » que nous ajouterons à notre chaîne « IA ».

Il devient évident que c'est là un processus récursif. Nous allons à nouveau isoler une à une les lettres de la chaîne restant à traiter (« M » puis « S » puis « O » puis « N ») que nous allons ajouter à la chaîne en cours, par exemple « IAO » suivi des anagrammes de « MSN ». Nous ferons ceci jusqu'à ce que la chaîne "à traiter" ne comporte plus qu'un seul caractère que nous ajouterons à la chaîne "en cours" et nous aurons une anagramme (Le nombre de caractères de la chaîne n'étant pas infini, nous sommes certains que la récursivité ne sera pas infinie).

La méthode pour générer des anagrammes recevra donc deux chaînes. La première sera la chaîne " en cours " et la seconde " restants ".

L'appel initial se fera avec une chaîne vide en 1er paramètre (il n'y a rien " en cours " et en second la chaîne complète.

Pour reprendre notre exemple ci-dessus, à un moment donné, après quelques appels, la méthode recevra "IA" et "MSON" en paramètres.

Le nombre d'appels imbriqués sera égal au nombre de lettre de la chaîne de départ moins 1. En effet, quand le second paramètre sera une chaîne ne comportant qu'un seul caractère, l'anagramme sera complète !

Avant de se lancer dans le code proprement dit, on peut donc imaginer sa structure…

Si la deuxième chaîne reçue ne contient qu'un seul caractère
- je l'ajoute à la première chaîne et j'ai une anagramme complète
- c'est fini
Sinon
- Dans une boucle
-- j'extrais une à une les lettres de la 2ème chaîne
-- j'ajoute cette lettre à la première chaîne reçue (qui était vide au 1er appel) : cela constituera ma nouvelle chaîne "EnCours"
-- Je prends les lettres restantes de la seconde chaîne et cela constitue ma nouvelle 2ème chaîne "restants"
-- J'appelle à nouveau la fonction avec les deux nouvelles chaînes que je viens de construire (et la récursivité se trouve ici :-)

```
`Méthode zAnagramme appelée par "zTestAnagrammes

C_ALPHA(20;$1;$2)
C_ALPHA(20;$Begin;$End)

$Begin:=$1
$End:=$2

$n:=Longueur($End)
```

```
Si ($n=1)  `anagramme prête

  $NewAnagram:=$Begin+$End
  POSITION  MESSAGE(2;2)
  MESSAGE($NewAnagram)

Sinon

  Boucle ($i;1;$n)

     `calcul de la nouvelle 1ère chaine
    $NewBegin:=$Begin+$End≤$i≥
     `calcul de la nouvelle 2ème chaine
    $NewEnd:=Sous  chaine($End;1;$i-1)+Sous  chaine($End;$i+1)
     `appel récursif de la méthode
    zAnagrammes ($NewBegin;$NewEnd)

  Fin  de  boucle
Fin  de  si
```

Optimisation de la méthode.

Comme vous le constatez, ma méthode comporte –au moins – deux défauts.

Tout d'abord, elle doit être appelée initialement par une autre méthode qui lui passe les paramètres. Cela pourrait être évité en utilisant la commande « nombre de paramètres ». En effet, si la méthode ne reçoit pas de paramètres c'est qu'il s'agit d'un appel initial. On peut donc, dans ce cas, demander à l'utilisateur de saisir une chaîne à ce moment là, puis rappeler la méthode en lui passant cette fois les paramètres. Cela ajoute un niveau de récursivité, mais cela permet de n'avoir plus qu'une seule méthode.

Ensuite, la méthode ne gère pas les lettres doubles ou triples. Le nombre d'anagrammes générées risque donc d'être trop important… La chaîne « AAAAAA » n'a qu'une seule anagramme !
Le nombre d'anagrammes d'une chaîne ayant une croissance de type factoriel (on y revient !) en fonction de son nombre de caractères, mieux vaut éviter les doublons… Le nombre exact d'anagrammes d'une chaîne est en fait égal à :

N= n ! /(x !*y !*z ! * …) ou "n" représente le nombre de caractères de la chaîne et ou x, y, z, etc représentent le nombre d'occurrences d'un même caractère.
Ex : la chaîne ABBCAADE possède 8 !/(2 !*3 !) anagrammes. (8 caractères, deux "B", trois "A").

Le plus simple pour traiter ce problème est donc de trier la chaîne avant de lancer le traitement, ce qui donne, en reprenant l'exemple ci-dessus : AAABBCDE. Le plus simple pour effectuer ce tri est d'alimenter un tableau de type Alpha(1) à partir de la chaîne initiale, de trier ce tableau (4D fait ça très bien tout seul:-), puis de reconstituer la chaine à partir des éléments du tableau trié.

Ensuite, lors de la boucle utilisant les caractères "restants", il ne faut considérer qu'une seule fois chaque caractère en le comparant avec celui qui l'a précédé et qui a été mémorisé (donc vide lors du premier test).

Pour reprendre l'exemple ci-dessous, une fois que l'on aura trouvé toutes les anagrammes commençant par la lettre "A", on passera directement à celles commençant par la lettre "B".

Ce qui nous donne le code suivant :

```
Si (Nombre de parametres=0)

 TABLEAU ALPHA(10;◊_Anag;1000)
 ◊ID:=0

  `pas de paramètre : on demande une chaine à l'utilisateur...
 $String:=Demander("Chaine de depart ? (6 caracteres Max) ")
 Si (ok=1) & ($String#"")
  $String:=Sous chaine($String;1;6)

   `on commence par trier la chaîne
   `c'est important pour éviter les doublons
   `pour cela on passe par un tableau
  $n:=Longueur($String)
  TABLEAU ALPHA(1;$_tempo;$n)
  Boucle ($i;1;$n)
   $_tempo{$i}:=$String≤$i≥
  Fin de boucle
  TRIER TABLEAU($_tempo;>)
  Boucle ($i;1;$n)
   $String≤$i≥:=$_tempo{$i}
  Fin de boucle
  TABLEAU ALPHA(1;$_tempo;0) `Efface le tableau

  TOUT SELECTIONNER([RESULTS])
  SUPPRIMER SELECTION([RESULTS])

  Anagrammes ("";$String) `on appelle la fonction avec une chaine triée

   `on enregistre ce qui n'a pas encore été stocké
   `(les anagrammes ont été stockées 1000 par 1000, on traite
   `les éléments du tableau non stockés)

  REDUIRE SELECTION([RESULTS];0)
  SUPPRIMER LIGNES(◊_Anag;◊ID+1;1000) `on supprime ce qui dépasse :-)
  TABLEAU VERS SELECTION(◊_Anag;[RESULTS]Value)

   `et voila, c'est fini.
  TOUT SELECTIONNER([RESULTS])
  VISUALISER SELECTION([RESULTS])

 Fin de si

Sinon

 C_ALPHA(20;$1;$2)
 C_ALPHA(20;$Begin;$End)

 $Begin:=$1
 $End:=$2
```

```
 $n:=Longueur($End)
S i ($n=1)  `anagramme prete
  $NewAnagram:=$Begin+$End
   ◊ID:=◊ID+1
   ◊_Anag{◊ID}:=$NewAnagram

  S i (◊ID>=1000)
   REDUIRE  SELECTION([RESULTS];0)
   TABLEAU  VERS  SELECTION(◊_Anag;[RESULTS]Value)
    ◊ID:=0
  Fin de si
 Sinon

  $Test:=""
  Boucle ($i;1;$n)
   S i ($End≤$i≥#$Test)
    $Test:=$End≤$i≥

    $NewBegin:=$Begin+$End≤$i≥
    $NewEnd:=Sous  chaine($End;1;$i-1)+Sous  chaine($End;$i+1)

    Anagrammes($NewBegin;$NewEnd)

   Fin de si
  Fin de boucle
 Fin de si


Fin de si
```

Exemple 3 : Une méthode devient un process !

Ce troisième exemple est un peu particulier, et ce, à deux points de vue.
1°) Il s'agit d'utiliser la même méthode pour créer un process et pour l'exécuter.
2°) C'est une méthode "pseudo" récursive.

Dans le second exemple, optimisé, nous avons vu que nous pouvions tester le nombre de paramètres afin de déterminer s'il s'agissait ou non du premier appel à une méthode. Dans l'exemple que nous allons maintenant aborder, nous allons faire de même et suivre le raisonnement suivant : Si la méthode ne reçoit pas de paramètre, cela signifie qu'elle est appelée à partir d'un process (pas forcément le process principal) et "tourne" donc à l'intérieur de ce process. Dans ce cas, nous allons utiliser la commande "NOUVEAU PROCESS" qui relancera cette même méthode en lui passant cette fois un paramètre. Ce paramètre n'a aucune espèce d'importance, hormis le fait d'exister. Lors de ce second appel, la méthode testera de la même façon la présence d'un paramètre. L'ayant reçu, elle ne lancera donc pas un nouveau process (Dans les faits, elle constitue déjà elle-même le nouveau process !).

Pourquoi est-ce "pseudo récursif" ?

La méthode ne s'appelle pas réellement elle-même, mais crée un nouveau process. Toute la différence est là. En effet, une fois exécutée, la méthode qui a lancé le process se terminera purement et simplement, et surtout

de façon indépendante de l'autre méthode, lancée par l'instruction "NOUVEAU PROCESS". Cela revient exactement au même que s'il s'agissait d'une méthode "A" qui lancerait une méthode "B" dans un nouveau process.

Le risque serait cette fois non plus de saturer la pile, mais de créer un process qui créerait lui même un process etc. et de saturer la mémoire à force de créer des process. Le résultat serait tout aussi catastrophique, mais la cause serait différente !

Une fois ce fonctionnement assimilé, on comprend que l'intérêt de ce type de programmation est essentiellement de limiter le nombre de méthodes dans une application, sans rendre le code trop difficile à lire. Des exemples de ce type de programmation se trouvent dans la base 4D Knowledge elle-même !

```
C_ENTIER  LONG($1)
C_ENTIER  LONG(◊PS_MonProcess)

Si (Nombre de parametres=0)

    `le process n'est pas encore lancé
 Si (◊PS_MonProcess=0)
   ◊PS_MonProcess:=Nouveau process("PS_MonProcess";64000;"PS_MonProcess";1)
 Sinon
  PASSER AU PREMIER PLAN(◊PS_MonProcess)
 Fin de si

Sinon
    `cette fois le process est lancé
    `METHODE PROCESS

    `FIN DE LA METHODE PROCESS

  ◊PS_MonProcess:=0

Fin de si
```

Conclusion

Ces exemples mettent en évidence une façon particulière de programmer. La récursivité fait parfois un peu peur. À tort, car bien employée, elle rend de fiers services.

Dans ces trois cas, nous n'avons pas jugé utile de « blinder » le code, car nous savions que le nombre d'appels récursifs serait limité. Dans le 1er cas : au nombre de lettres du mot fourni, dans le deuxième au nombre d'imbrications de dossiers au sein d'un volume, et dans le troisième à ... zéro. Les risques de saturation de piles sont minimes, voire nuls.

En revanche, d'autres cas peuvent se présenter où le contrôle du nombre de niveaux devra impérativement être effectué. Cela se fait facilement à l'aide d'une variable process ou interprocess que l'on incrémente avant chaque appel de la méthode récursive et qui détermine l'opportunité de l'appel ou non. De plus la commande « AP_Available memory » qui fait partie du plugin 4D Pack, pourra également être mise à profit pour éviter tout désagrément concernant la mémoire disponible…