Penser récursif

Dominique Michelucci, Université de Dijon

23 septembre 2012

La *récursion* est une notion fondamentale : selon la "thèse de Kleene", seules sont calculables les fonctions *récursives* ; la théorie de la complexité définit des ensembles *récursifs*, et des ensembles *récursivement* énumérables ; les équations donnant la complexité d'un algorithme (comme $T(1) = 1, T(n) = 2T(n) + O(n) \Rightarrow T(n) = O(n \log n)$ pour le tri fusion ou la transformée de Fourier rapide) sont *récursives*.

Recettes pour penser récursif :

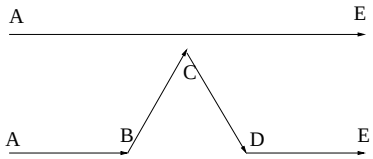
Traitez d'abord les cas triviaux (dits terminaux). Ensuite réduisez le cas général, et non trivial, à un cas ou des cas plus simples, pas forcément triviaux. Si vous avez réduit le cas général, même un tout petit peu, vous pouvez utiliser la méthode que vous êtes en train de concevoir, et qui n'est pas encore terminée, pour résoudre le cas réduit. N'est ce pas magnifique ?

Dans ce contexte (la conception d'une méthode), ne cherchez surtout pas à déployer l'arbre des appels récursifs ! C'est inutile, et même nuisible, car cela complique (au lieu de simplifier...) et va vous noyer sous une avalanche de calculs sans intérêt ; il faut, au contraire, vous concentrer sur l'essentiel, qui est : pouvez vous réduire le problème à un (ou des) problèmes plus simples, et combiner les solutions de ces problèmes plus simples pour générer la solution de votre problème ?

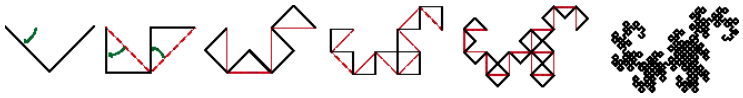
Par contre, si vous vous intéressez à la complexité d'une méthode récursive, vous pouvez en effet déployer l'arbre des appels récursifs – sur de petits exemples !. Cela peut aussi vous suggérer des améliorations de votre méthode. Une méthode d'optimisation classique est la mémorisation, ou *memoization*, présentée plus bas.

Voici quelques exercices pour vous entrainer à la récursion – ou vous convaincre que c'est plus facile en récursif !

Exercice 1 : écrivez un algorithme pour dessiner le fractal de Von Koch. Chaque étape remplace le segment AE par 4 autres segments, comme sur cette figure :



Exercice 2 : écrivez un algorithme pour dessiner la courbe du dragon.



Exercice 3 : Ecrire la méthode calculant $C(k, n)$, le nombre de façons de choisir k objets parmi n.

Exercice 4a : un arbre binaire, ordonné, est donné : pour tous ses noeuds, les valeurs éventuelles dans le fils gauche sont inférieures ou égales à celle du noeud, qui est inférieure ou égale à toutes les valeurs dans le fils droit. Vous pouvez le supposer équilibré. Écrire un algorithme pour trouver la plus grande valeur dans l'arbre qui soit inférieure à un nombre donné s. S'il n'y en a pas (par exemple l'arbre est vide, ou toutes les valeurs dans l'arbre sont plus grandes que s), alors par convention votre méthode rendra ∞.

Exercice 4b : même question que 4a, mais proposez une méthode non récursive.

Exercice 5 : Ecrire un tri rapide (*quicksort*) non récursif. C'est plus difficile ! Vous aurez besoin d'une pile, ou d'une autre structure d'attente.

Exercice 6 : Ecrire un tri par fusion (*mergesort*) non récursif.

Exercice 7 : Tours de Hanoi.

Exercice 8 : lister les éléments d'un arbre binaire (ordonné) du plus petit au plus grand.

Exercice 9 : La mémorisation (memoization pour les informaticiens anglo-saxons), ou l'"attribut-fonction", permettent d'optimiser les méthodes récursives ; les valeurs calculées sont stockées dans une table de hachage, ou attachées en attribut sur les données pour la méthode de l'"attribut-fonction" (cette dernière est utilisable quand la fonction ne prend qu'un seul paramètre). Il est aussi possible d'utiliser des pointeurs faibles (si le langage le permet) : cela permet au gestionnaire de mémoire (le *garbage collector*) de récupérer l'espace mémoire occupé par des données qui n'ont pas été utilisées depuis "longtemps". Cette méthode est intéréssante dans les cas de récursion "tordue". Re-écrivez l'algorithme naïf pour calculer Fibonacci avec cette m'ethode ; que devient la complexité ? Dans le cas général (pas Fibonacci), pourquoi conseille-t-on d'utiliser une table de hachage, et pas un tableau ?

Exercice 10. La récursion permet de définir des fonctions de $\mathbb{N}$ vers $\mathbb{N}$ qui croissent très rapidement. La fonction d'Ackermann, ou la fonction de Kozen, en sont deux exemples. La fonction de Kozen est définie par : $K_0(x) = x+1, K_{n+1}(x) = K_n^{(x)}(x)$, où la notation $f^{(k)}$ signifie que f est appliquée k fois : $f^{(0)}(x) = x, f^{(1)}(x) = f(x), f^{(2)}(x) = f(f(x)), f^{(3)}(x) = f(f(f(x)))$. Prouvez que $K_1(x) = 2x$, que $K_2(x) = x2^x$. Calculez $K_3(2), K_3(3)$. Il est possible de définir des fonctions encore pire : $G_0(x) = x + 1, G_{n+1}(x) = G_n^{(G_n(x))}(x)$: tentez de calculer $G_3(1)$ Ces exemples montrent que certaines fonctions récursives (donc parfaitement définies) sont incalculables en pratique.