RECURSION

- Idée très simple: une fonction peut faire appel à elle-même
  exactement comme un appel d'une autre fonction: dès l'appel récursif achevé l'exécution du programme continue (et sans effet sur les variables et paramètres de l'appel initial)

- exemples favoris: factoriels, nombres de Fibonacci

- valable pour fonctions et procédures (fonctions `void`)

- Aussi récursion mutuelle: deux ou plusieurs fonctions s'appellent entre elles (donc appel avant déclaration pour quelques unes)

LES EXEMPLES FAVORIS SONT MAUVAIS

- la fonction récursive est une méthode bizarre de calculer factoriel
  une boucle simple suffit

- la fonction récursive est une très mauvaise méthode de calculer Fibo
  le temps de calcul est exponentiel!

- Peut-être une méthode récursive est plus élégante et claire

- Mais on doit aussi considérer l'efficacité

MEILLEURS EXEMPLES

- x^n par calcul de carrés: on saute de x^i à x^{2i} dans une seule multiplication; plus difficile à gérer avec une boucle

- Les tours de Hanoi: déplacer une tour de n disques de taille différente d'une colonne à une autre en utilisant une seule colonne auxiliaire, selon la règle qu'on ne peut déplacer qu'un disque à la fois et que chaque colonne a toujours ses disques empilés en ordre décroissant de taille

- Affichage en (disons) binaire
  une boucle simple affiche les chiffres dans l'ordre inversé

- Boucles imbriquées de profondeur inconnue

- Algorithmes "diviser pour régner"

PARAMETRES ET VARIABLES LOCALES

Les paramètres et les variables déclarées localement dans une fonction (si elle est récursive ou non)

- sont créés au moment de l'appel/activation de la fonction

- continuent à exister jusqu'a la sortie finale de l'appel

- sont inaccessibles et immuables pendant d'autres appels imbriqués (sauf utilisation de pointeurs...)

Donc, quand une fonction s'appelle récursivement, à un moment de l'exécution du programme, il existe un nombre indéfini d'occurrences de ses paramètres et variables, mais, en général une seule occurrence accessible.

Et possibilité d'échec parce que l'espace mémoire ne suffit pas pour toutes ces variables ("segmentation fault" en C)

EXEMPLES UTILISANT DES TABLEAUX

- Toutes les listes multiplicatives avec terminaison n

- Recherche dichotomique

- recherche d'un parcours de l'échiquier par un cavalier

- le même avec utilisation intelligente du nombre de voisins toujours à visiter pour chaque case

  – élaguer: si le nombre est nul il faut y aller (sauf la case de départ)

- heuristique: de préférence aller au voisin avec le nombre le plus bas

Eviter les effets de bord (mais aussi les copies coûteuses)!

FONCTIONS PURES

- Théorie selon laquelle les fonctions écrites sans affectation et effet de bord sont plus claires et plus facilement écrites sans bugs
- plus proche de l'idée mathématique d'une fonction; et les définitions ressemblent à des équations
- Une question de goût
- mais récursion nécessaire pour ce style de programmation

QUAND DOIT ON L'UTILISER?

- Quels sont les fonctions pour lesquelles la récursion est essentielle?

- Aucune: ce qu'on peut faire avec récursion, on peut toujours faire avec une pile explicite!

- Jeux de parcourir un arbre sans récursion et sans pile!!

QUAND DEVRAIT ON L'UTILISER?

- Quand le code est plus simple à écrire et comprendre avec récursion

- Dont presque tous les cas où l'alternative serait une pile explicite

- Mais si je veux parcourir deux arbres en même temps (par exemple je veux les fusionner), je ne peux pas forcément le faire récursivement
(mais "coroutines" récursives existent en certains langages)

LE COUT

- Espace: négligeable:
  nécessité de stocker les paramètres et variables locales de chaque appel (activation); utilisation invisible d'une pile; sauf très gros tableaux, cet espace n'est pas important

- Temps: négligeable:
  les instructions supplémentaires exécutées à chaque appel d'une fonction prennent un temps petit par rapport aux calculs de la fonction sauf si ces calculs sont simples

- La qualité du programme est plus importante
  et un bon compilateur peut quelquefois générer le même code pour les deux versions (cas de récursion à la fin)