Table des matières

A. Fonctions récursives

1. Qu'est ce que la récursivité ?

Une notion est dite récursive lorsqu'elle se contient elle-même en partie ou si elle est partiellement définie à partir d'elle-même. La récursivité est appuyée sur le raisonnement par récurrence. Typiquement il s'agit d'une suite dont le terme général s'exprime à partir de termes qui le précèdent. Par exemple la factorielle d'un nombre N donné est le produit des nombres entiers inférieurs ou égaux à ce nombre N. C'est noté N! avec par définition la factorielle de 0 à 1, ce qui donne :

0! = 1

1! = 1

2! = 1*2

3! =1*2*3

(…)

N! = 1*2*3 ….*(N-1)*N

La notation générale est :

N! = 1 si N = 0

N! = N*(N-1)! si N > 0

et l'on voit que la factorielle de N est définit en fonction d'elle-même (N-1)!, c'est un processus récursif.

2. Une fonction récursive basique

Une fonction récursive est ainsi tout simplement une fonction qui s'appelle elle-même. De ce fait un algorithme récursif va jouer sur les paramètres en entrée de la fonction qui seront modifiés à chaque nouvel appel de la fonction dans son propre corps. Par exemple dans un tri au départ nous avons un ensemble D et la récursion s'exerce sur des sous-ensembles de D jusqu'à ce qu'il n'y ait de sous ensemble possible.

L'appel de la fonction par elle_même peut être directe, par exemple une fonction f() qui appelle f(), ou indirecte si f() appelle g() qui appelle f(). Nous nous en tiendrons à la récursion directe dans la suite de ce module.

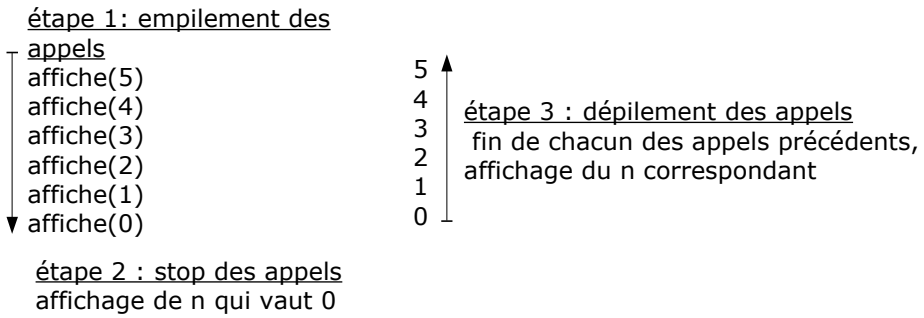
Pour une fonction récursive il y a trois aspects fondamentaux :

- le test d'arrêt : comment s'arrête la succession des appels ?
- la décomposition des ou de la donnée(s) : sur quelle base sont faits les appels successifs ?
- l'empilement puis le dépilement en mémoire des appels successifs. Chaque appel possède en effet ses propres variables locales en mémoire (comme s'il s'agissait d'appels de fonctions différentes)

Voici un exemple de fonction récursive, qu'imprime le programme suivant ?

```
void affiche(int n)
{
   if (n>0)
      affiche(n-1);
   printf("%d ",n);
}
int main()
{
   affiche(5);
   return 0;
}
```

Le processus peut s'exprimer ainsi : tant que n est supérieur à 0, affiche( ) est appelée avec avec n-1 en paramètre. Le processus complet peut être décomposé en trois étapes :



Ainsi le programme imprime :

```
0 1 2 3 4 5
```

Maintenant si la fonction est modifiée comme suit :

```
void affiche(int n)
{
   printf("%d ",n);          // affichage au début
   if (n>0)
      affiche(n-1);
}
```

Le programme imprime alors la suite décroissante :

```
5 4 3 2 1 0
```

En effet, l'affichage à l'écran à lieu avant l'appel récursif, n est d'abord imprimé ensuite l'appel a lieu sur n-1.

Si printf( ) a lieu au début et à la fin de la fonction :

```
void affiche(int n)
{
   printf("%d ",n);          // affichage au début
   if (n>0)
      affiche(n-1);
   printf("%d ",n);          // et à la fin
}
```

La suite qui apparaît est :

```
5 4 3 2 1 0 0 1 2 3 4 5
```

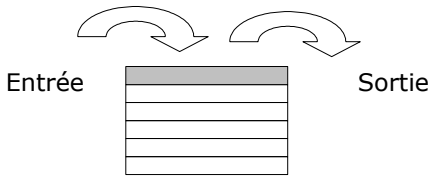
A travers cet exemple très simple le lien entre itération (boucle) et récursivité est évident, la version itérative de la fonction affiche() est :

```
 void affiche(int n)
{
int i;
   for (i=0; i<=n; i++)
      printf("%d ",i);
}
```

Un processus récursif peut théoriquement toujours être remplacé par un processus itératif associé à la gestion d'une pile. Toutefois les deux ne sont pas identiques et il existe des fonctions "authentiquement récursives" c'est à dire très compliquées à remplacer par des processus itératifs. Il y a des différences de rapidité à l'exécution. Selon les cas, c'est l'un ou l'autre le plus rapide et il n'est pas possible de généraliser.

3. Pile d'appels et débordement

Lorsqu'un appel de fonction est réalisé, un espace mémoire est alloué pour toutes les données de la fonction. Cet espace mémoire est libéré au moment à la fin de l'exécution de la fonction, lors du retour au contexte d'appel. Tous les appels de fonctions sont stockés dans une pile en mémoire. Un pile fonctionne comme une pile d'assiettes lorsqu'on fait la vaisselle : les assiettes à laver sont empilées les unes par dessus les autres au fur et à mesure de leur arrivée et la personne qui lave les prends en commençant par le haut, ce qui fait que la dernière arrivée est la première lavée, last in first out (LIFO) :



De même chaque fonction appelée est empilée et dépilée à la fin de son exécution. Ainsi lorsqu'un programme est actif la fonction main() est tout en bas de la pile et au sommet se trouve la fonction en train de s'exécuter. La pile des appels de fonction (call stack en anglais) a une taille maximum qu'il n'est pas possible à la machine de dépasser. L'appel de trop provoque un débordement de pile qui en général met fin à l'exécution du programme avec un message d'erreur de la part du système d'exploitation.

C'est un problème qui peut se poser avec des fonctions récursives lorsque le nombre d'appels récursifs est trop élevé. Le programme suivant en fait la démonstration. La fonction récursive deborde() n'a pas de test d'arrêt, elle va donc s'appeler à l'infini jusqu'à ce que le système mette fin au programme suite à un débordement de pile. Les appels sont comptés via une variable passée par référence et le numéro d'appel est tout d'abord affiché :

```
#include <stdio.h>
```

```
#include <stdlib.h>

void deborde (unsigned int*cmpt)
{
   printf("%d\n",(*cmpt)++);
   deborde(cmpt);
}

int main()
{
int cmpt=0
          deborde(&cmpt);
}
```

Sur ma machine je constate qu'il y a 130160 appels avant débordement.

Remarque

Le nombre d'appels récursifs d'une fonction récursive lors de son exécution donne ce que l'on appelle *la profondeur de la récursion*. C'est la différence à un moment donné de l'exécution entre le nombre d'appels engagés depuis le début et le nombre de retours effectués.

4. Retourner une valeur

La gestion du mécanisme de retour avec une fonction récursive nécessite de faire un peu attention. Soit une fonction qui additionne tous les nombres de 0 à n et retourne le résultat. Par exemple avec n=5 l'objectif est d'obtenir 0+1+2+3+4+5 et de retourner 15.

```
#include <stdio.h>
#include <stdlib.h>

int add(int n)
{
int res=n;
   if (n>0){
     res+=add(n-1);
   }
   return res;
}

int main()
{
   printf("res=%d",add(5));
   return 0;
}
```

L'écriture peut être simplifiée, en effet la variable res finalement ne fait que doubler la variable n, celle-ci peut être augmentée de la valeur précédente à chaque retour et finalement donner le résultat souhaité :

```
int add(int n)
{
   if (n>0){
      n+=add(n-1);
   }
   return n;
```

```
}
```

Cette écriture peut encore être affinée en utilisant l'opérateur conditionnel

```
(test) ? val si vrai  :  val si faux ;
```

```
int add(int n)
{
   return (n>0) ? n + add(n-1) : 0;
}
```

Autre exemple, compter le nombre de soustraction d'une valeur a d'un nombre n, tant que n reste positif (ce qui revient à une division entière de n par a)

```
int cmpt(int n, int a)
{
   if (n<a)
      return 0;
   return 1 + cmpt(n-a, a);
}
```

ce qui peut se réduire à

```
int cmpt(int n, int a)
{
   return n < a ? 0 : 1 + cmpt(n-a, a);
}
```

Le résultat de la fonction cmpt() est obtenu lors du dépilement, à chaque dépilement 1 est ajouté à la valeur retournée.

Mais le compte de cette fonction pourrait être opéré à l'empilement, moyennant une variable supplémentaire. Soit une variable globale, soit une variable passée par référence, soit une variable locale déclarée en statique :

version avec variable globale :

```
int res=0;

void cmpt(int n, int a)
{
  if (n>=a){
     res++;
     cmpt(n-a,a);
  }
}
```

Version avec passage par référence :

```
void cmpt(int n, int a,int*res)
{
  if (n>=a){
     (*res)++;
     cmpt(n-a,a,res);
  }
}
```

Version avec variable statique :

```
int cmpt(int n, int a)
{
static int res=0;
```

```
  if (n>=a){
     res++;
     cmpt(n-a,a);
  }
  return res;
}
```

5. Se représenter et analyser le fonctionnement

A partir du moment où il y a plusieurs appels récursifs de la fonction au sein de la fonction il devient difficile de se représenter clairement son fonctionnement. Par exemple, qu'imprime le programme suivant ?

```
#include <stdio.h>
#include <stdlib.h>

void p(int n)
{
   if (n>0){
      p(n-2);
      printf("%3d",n);
      p(n-1);
   }
}

int main()
{
   p(4);
   return 0;
}
```

Il y a deux façons d'analyser le fonctionnement d'une fonction de ce type, l'analyse descendante et l'analyse ascendante.

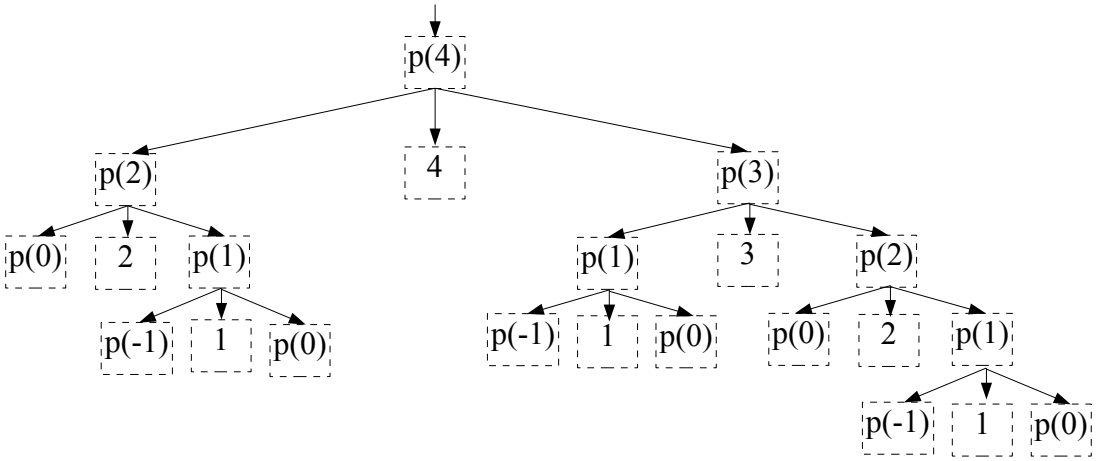
a. Analyse descendante

On commence par étudier le premier appel qui a lieu dans le main() : p(4)

Avec la valeur 4 ça donne pour la suite :

```
p(2); printf("%3d",4); p(3);
```

Déjà il est prévisible que le nombre 4 sera donné en sortie à un moment donné. Faisons de même pour les appels p(2) et p(3) et les suivants... la représentation va prendre l'allure d'un arbre :



Les appels ont lieu si n>0, il n'y a donc pas d'appel pour n=-1 et n=0. A chaque fois qu'il n'y a plus d'appel récursif la suite des instructions de l'appel courant est effectuée, c'est à dire l'affichage de la valeur de n. L'arbre va se lire de gauche à droite : il revient pour afficher n après chaque appel à gauche de p(0) ou p(-1) ce qui donne :

```
2 1 4 1 3 2 1
```

Avec cette méthode on voit que des sous-arbres se répètent, ce sont les appels p(1).

b. Analyse ascendante

Cette fois nous allons partir à l'inverse, par le bas avec le dernier appel possible lorsque n=1 et nous allons remonter ensuite pour la valeur immédiatement supérieure ce qui donne :

p(1) : 1

p(2) : p(0) 2 p(1) soit 2 1

p(3) : p(1) 3 p(2) soit 1 3 2 1

p(4) : p(2) 4 p(3) soit 2 1 4 1 3 2 1

6. Choisir entre itératif ou récursif

Soit la fonction récursive suivante qui propose d'additionner des nombres entrés par l'utilisateur :

```
#include <stdio.h>
#include <stdlib.h>

void somme(int*s)
{
int n;
   if (scanf("%d",&n)==1){
      *s+=n;
      somme(s);
   }
}
```

```
int main()
{
int res=0;
   printf("entrer des nombres, une lettre pour arrêter  :\n");
   somme(&res);
   printf("resultat : %d\n",res);
   return 0;
}
```

L'appel récursif a lieu juste à la fin de la fonction, il est très facile dans ce cas d'avoir un processus itératif et la récursion n'a pas vraiment lieu d'être :

```
void somme(int*s)
{
int n;
   while (scanf("%d",&n)==1)
      *s+=n;
}
```

D'une façon générale, la récursion est nécessaire lorsque la fonction récursive s'appelle elle-même plusieurs fois (comme la fonction p() en 1.5) par exemple lorsqu'il s'agit de parcourir des figures complexes comme les arbres, des graphes, ou encore pour rechercher une zone dans une matrice en partant dans toutes les directions, comme il faut le faire pour un démineur par exemple.

B. Exemples classiques de fonctions récursives

Voici un ensemble d'exemples souvent cités et bien intéressants pour s'approprier la récursivité et l'écriture de fonctions récurisives.

1. 1. Calculs

a. Afficher les chiffres d'un entier

Soit un entier de valeur 45671, il s'agit d'afficher successivement les caractères 4, 5, 6, 7, 1 et non plus la valeur de l'entier. Pour ce faire, tant que le résultat est supérieur à 0 on prend le modulo 10 qui donne le dernier chiffre, on divise par 10 et on recommence, en itératif ça donne :

```
void chiffreIter (int val)
{
   while (val>0){
       printf("%d_", val%10);
       val/=10;
   }
}
```

La version récursive est très simple il suffit de remplacer la boucle par un appel récursif :

```
void chiffre(int val)
{
   if (val>0){
      printf("%d_", val%10);
```

```
      chiffre(val/10);
   }
}
```

b. Factorielle

Un produit factoriel c'est : n! = 1*2*3*...*n. La définition c'est :

n! = 1 si n=0

n! = n(n-1) si n>0

Il y a une fonction de récurrence et on boucle sur n tant que n>0, voici une version de fonction itérative :

```
int factIter (int n)
{
int res=1;
   while (n>1)
      res*=n--;  // attention décrémentation après multiplication
   return res;
}
```

D'une certaine façon la version récursive peut sembler plus facile, plus proche de la définition du calcul. si n est égal à 0 la fonction retourne 1, sinon la fonction retourne n*(n-1) ce qui donne :

```
int fact(int n)
{
   if (n==0)
      return 1;
   else
      return n*fact(n-1);
}
```

Éventuellement l'écriture peut être rendue plus concise avec l'opérateur conditionnel

```
int fact(int n)
{
   return n>1 ? n*fact(n-1) : 1;
}
```

c. Suite de Fibonacci

Il s'agit d'une suite établie comme suit :

Pour n >= 0

fib(n) = n si n = 0 ou n = 1

fib(n) = fib(n-2) + fib(n-1) si n > 1

Par exemple pour les valeurs de 0 à 8 nous avons :

n	0	1	2	3	4	5	6	7	8
fib(n)	0	1	1	2	3	5	8	13	21

Voici une version itérative de la fonction :

```
int fibIter(int n)
{
int i,res,f1,f2;

   if (n<=1)
      res=n;
   else{
      f2=0;
      f1=1;
      for (i=2; i<=n; i++){
         res=f2+f1;
         f2=f1;
         f1=res;
      }
   }
   return res;
}
```

La version récursive est finalement plus naturelle,

```
int fib(int n)
{
   if (n<=1)
      return n;
   else
      return fib(n-2)+fib(n-1);
}
```

L'opérateur conditionnel peut rendre encore plus concise l'écriture :

```
int fib(int n)
{
   return n<=1 ? n : fib(n-2)+fib(n-1);
}
```

d. Changement de base arithmétique d'un nombre

De base 10 vers base B pour un nombre positif et avec 0 < B <=16

A chaque étape on prend le modulo du nombre en base 10 selon la base B donnée puis on divise le nombre en base 10 par la base B. Pour la version itérative le résultat obtenu commence par la plus petite unité : il est donné à l'envers. Par exemple pour 160 en base dix passé en base 8 on obtient successivement 0, 4, 2 et il faut lire le résultat en commençant par la fin : 240. La solution consiste à empiler le résultat obtenu à chaque étape du calcul et lorsque le calcul est terminé de dépiler pour afficher dans le bon ordre chaque chiffre obtenu, ce qui donne :

```
void conversionIter (int val, int base)
{
int pile[100];
int sommet=0;
   while (val>0){
       pile[sommet++] = val % base;
       val /= base;
   }
   for (--sommet; sommet>=0; sommet--)
```

```
      if(pile[sommet]<=9)
         printf("%d",pile[sommet]);
      else
         printf("%c",pile[sommet]-10+'A');
}
```

La version récursive évite la gestion de la pile du fait de l'empilement des appels successifs :

```
void conversion (int val, int base)
{
static char chif[]="0123456789ABCDEF";  // de 0 à 15

   if (val/base != 0)
      conversion(val/base,base);
   printf("%c",chif[val%base]);   // vaut entre 0 et 15 compris
}
```

De base B vers base 10 pour un nombre positif et sans lettre pour l'hexadécimal

Le principe est d'utiliser la notation étendue qui permet de décomposer un nombre. Par exemple 123 en base 8 donne en base 10 l'addition :

$1*8^2 + 2*8^1 + 3*8^0 = 83$

La version itérative calcul chaque étape en commençant par la droite et les ajoute une par une. Ainsi $3*8^0$ s'obtient en faisant (123%10)*1 (un nombre à la puissance 0 vaut toujours 1)

ensuite :

$2*8^1$ s'obtient avec ((123/10)%10)*8
$1*8^2$ s'obtient avec ((12/10)%10)*8*8

A chaque tour le nombre obtenu est additionné au résultat précédent.

Le principe est toujours le même quelque soit au départ le nombre et sa base.

Pour l'algorithme il est important de dissocier la division par 10 du reste obtenu avec modulo 10. Au départ le résultat c'est un modulo 10. Ensuite, à chaque tour la valeur donnée est divisée par 10 tant qu'elle est supérieure à 0. A chaque tour également est pris le modulo 10 de la nouvelle valeur obtenue. La base est multipliée par elle-même et le résultat stocké au fur et à mesure dans une variable b. Le produit de chaque tour est ajouté au résultat final, ce qui donne :

```
int convert10Iter (int val, int base)
{
int res,b,r;
   b=1;
   res=val%10;
   while (val>0){
      val=val/10;
      r=val%10;
      b*=base;
      res+=r*b;
   }
   return res;
}
```

La version récursive est un peu plus concise :

```
int convert10(int val, int base)
```

```
{
   if (val/10 ==0)
      return val%10;
   else
      return convert10(val/10,base)* base + val % 10;
}
```

Puissance

Pour calculer x^n on multiplie n fois x par lui-même, par exemple 3^4=3*3*3*3. La fonction récursive correspondante donne :

```
int puissance(int x, int n)
{
   if (n==0)
      return 1;
   else
      return x*puissance(x,n-1);
}
```

ou en utilisant l'opérateur conditionnel :

```
int puissance(int x, int n)
{
   return (n==0) ? 1 :x*puissance(x,n-1);
}
```

Le temps de calcul peut être diminué en ne calculant que la moitié puis en multipliant cette moitié par elle-même, par exemple pour 3^4 faire simplement 3^2 puis 3^2 * $3^{2.}$ Dans le cas d'un exposant impair il suffit de multiplier une fois de plus par la base x, par exemple :

$3^5 = 3^2 * 3^{2.} * 3$

Ainsi pour x^n le calcul se ramène à trouver $x^{n/2}$, le multiplier par lui-même et si n est impair multiplier une fois par x en plus. Ce principe peut être implémenté de façon récursive de la façon suivante :

```
int puissance2(int x, int n)
{
int res;
   if (n==0)
      res = 1;
   else{
      res=puissance2(x, n/2);
      if(n%2==0) // si pair
         res=res*res;
      else
         res=res*res*x;
   }
   return res;
}
```

PGCD, algorithme d'Euclide

Soit deux nombres entiers x et y dont un au moins n'est pas nul, *pgcd(x,y),* c'est le nombre entier le plus grand par lequel x et y sont divisibles. Par exemple :

```
pgcd(1000,600)   =  200
pgcd(-70,90)     = 10
pgcd(0,12)       = 12
```

L'algorithme repose sur quelques remarques :

```
1) pgcd (x,y) = pgcd (y,x)

2) Si d divise x et y, d divise aussi x - y  soit la propriété :
pgcd (x,y) = pgcd (x-y,x)

3) pcgd(x,0) = 0
```

Le principe de l'algorithme est de soustraire du plus grand des deux nombres le plus petit. Par exemple :

```
pgcd(1900,700)        = pgcd(1200,700)          // remarque 2
                      =  pgcd(500,700)          // remarque 2
                      =  pgcd(700,500)          // 1
                      =  pgcd(200,500)          // 2
                      =  pgcd(500,200)          // 1
                      =  pgcd(300,200)          // 2
                      =  pgcd(100,200)          // 1
                      =  pgcd(200,100)          // 2
                      =  pgcd(100,100)          // 2
                      =  pgcd(0,100)            // 2
                      =  pgcd(100,0)            // 1
                      = 100                     // 3
```

En fait le résultat est obtenu lorsque x et y sont égaux. Ainsi, tant que x et y sont différents, à chaque tour : si x est inférieur à y on permute x et y, et ensuite on soustrait y de x, ce qui donne :

```
int pgcd(int x, int y)
{
int t;
   while (x != y){
      if (x < y){
         t = x;
         x = y;
         y = t;
      }
      x -= y;
   }
   return x;
}
```

Attention cet algorithme ne prend pas en compte des nombres négatifs ni le fait que dés le départ x ou y peuvent être égaux à 0.

Pour diminuer le nombre de tours il est possible d'utiliser l'opérateur modulo plutôt que de faire une soustraction. En effet, x%y vaut le reste de la division de x par y, par exemple 1900%700 vaut directement 500 sans avoir à faire 1900-700 et 1200-700.

On remarque également que l'expression x%y est nécessairement inférieure à y. Alors plutôt que d'écrire :

```
pgcd(x,y) = pgcd( x%y, y)
```

puis de permuter on peut écrire directement :

```
pgcd(x,y) = pgcd( y, x%y)
```

et appliquer cette règle tant que y est différent de 0.

Par exemple pgcd(1900,700) ne prend plus que 5 étapes :

```
pgcd(1900,700)        =  pgcd(700,500)
                      =  pgcd(500,200)
                      =  pgcd(200,100)
                      =  pgcd(100,0)
                      =  100
```

Tant que y est supérieur à 0 on calcul m = x % y, ensuite on affecte y à x et on met m dans y, ce qui donne en itératif quatre instructions :

```
int pgcd(int x, int y)
{
int ;
   while(y){
      m=x%y;
      x=y;
      y=m;
   }
   return x;
}
```

La version récursive permet de permuter directement les paramètres (y passé en x et x%y passé en y) du coup la fonction qui en découle est simplifiée :

```
int pgcd(int x, int y)
{
   if (y)
      return pgcd(y,x%y);
   else
      return x;
}
```

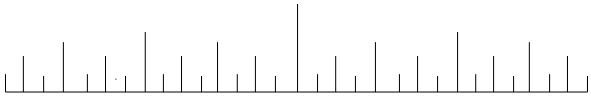
Et une seule ligne avec l'opérateur conditionnel :

```
int pgcd(int x, int y)
{
   return (y) ? pgcd(y,x%y) : x;
}
```

2. Dessins

a. Tracé d'une règle graduée : "diviser pour résoudre"

Lors d'un appel récursif, la fonction reste la même mais les valeurs passées aux paramètres changent. L'objectif est ici de tracer une règle graduée du type :



la graduation est marquée par des marques régulières, pour chaque section il y a une marque aux moitiés de section, une marque plus petite aux quarts de section, une marque encore plus petite aux huitièmes de section etc. Cet exercice illustre la technique récursive du "diviser pour résoudre" souvent utiliser dans des problématiques d'optimisation et des algorithmes de tris. Les données sont divisées et ne se recouvrent jamais (on ne repasse pas deux fois au même endroit).

La graduation est obtenue en traçant un trait de hauteur h toujours à la moitié m du segment courant et récursivement de tracer un trait de hauteur h-1 à la moitié m/2 du segment précédent. On suppose une fonction trace() pour tracer le trait verticale, cette fonction dépend de l'environnement. Elle prend en paramètre, la position horizontale du trait et sa hauteur. Ici nous utilisons très simplement, un affichage console du nombre correspondant à la hauteur à la bonne place horizontalement. De plus chaque hauteur a sa propre couleur pour faciliter la lecture du résultat. La fonction nécessite les fonctions gotoxy() et textcolor() données en annexe. Ce qui donne :

```
void trace(int hpos, int h)
{
   gotoxy(hpos,0);
   textcolor(15-h);
   printf("%d",h);
}
```

Les graduations de la règle sont obtenues avec la fonction :

```
void regle(int gauche, int droite, int hauteur)
{
int milieu=(gauche+droite)/2;

   if (hauteur>0){
      trace(milieu,hauteur);
      regle(gauche,milieu,hauteur-1);
      regle(milieu,droite,hauteur-1);
   }
}
```

Si hauteur n'est pas nulle, la fonction commence par tracer un trait de "hauteur"au milieu du segment [gauche,droite] puis elle s'appelle deux fois pour "hauteur-1" , une fois pour le segment [gauche,milieu] et une fois pour le segment [milieu,droite].

Prenons par exemple un segment de 0 à 8 et une hauteur de 3, au départ nous avons l'appel :

```
regle( 0, 8, 3)  ensuite :
    tracer (4,3)
    regle(0,4,2)
        tracer (2,2)
        regle (0,2,1)
            tracer (1,1)
            regle (0,1,0)
            regle (1,2,0)
        regle (2,4,1)
            tracer (3,1)
            regle (2,3,0)
            regle (3,4,0)
    regle (4,8,2)
        tracer (6,2)
        regle (4,6,1)
            tracer (5,1)
            regle (4,5,0)
            regle (5,6,0)
        regle (6,8,1)
            tracer (7,1)
            regle (6,7,0)
            regle (7,8,0)
```

Avec notre fonction trace() ci-dessus nous obtenons pour cet appel :

1 *2* 1 **3** 1 *2* 1

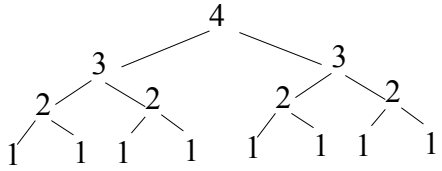
Avec les appels :

regle(0,16,4) nous obtenons la suite : 1 *2* 1 **3** 1 *2* 1 **4** 1 *2* 1 **3** 1 *2* 1

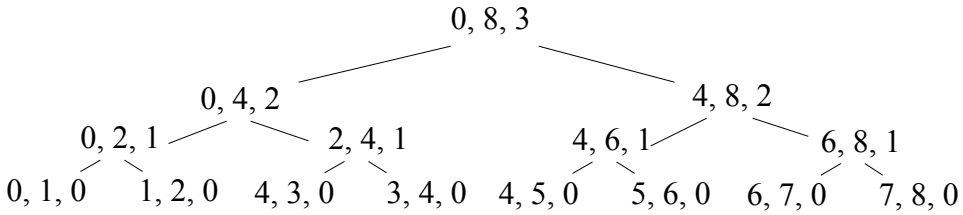
regle(0,32,5) nous obtenons :

1 *2* 1 **3** 1 *2* 1 **4** 1 *2* 1 **3** 1 *2* 1 5 1 *2* 1 **3** 1 *2* 1 **4** 1 *2* 1 **3** 1 *2* 1

On distingue du point de vue des appels comme du résultat la forme d'un arbre :



Cet arbre sous-jacent correspond à l'arbre d'appels. Voici par exemple l'arbre d'appels pour regle(0,8,3), il reprend le détail de la reconstitution des appels ci-dessus ans indiquer l'ordre chronologique du parcours :



Il s'agit d'un arbre binaire complet : à chaque nœud de l'arbre partent deux branches jusqu'aux extrémités, appelées les feuilles de l'arbre. Le premier nœud est appelé la racine de l'arbre.

Le nombre total de nœuds d'un arbre binaire complet a rapport avec sa hauteur h : c'est $2^h - 1$.

En l'occurrence c'est pratique ici pour obtenir la taille de notre règle en fonction de la hauteur qu'on lui donne : pour une hauteur de 3 il faut 2^3-1 = 7 graduations, pour une hauteur de 4 il faut 2^4-1 = 15 graduations, pour une hauteur de 5, 2^5-1 = 31 etc.

Le nombre total de feuilles correspond lui à 2^h, le nombre de feuilles par niveau correspond à 2^{niveau} avec 0 pour le niveau racine.

La version itérative de la règle consiste à tracer chaque niveau de l'arbre en commençant par celui du bas et en les superposant au fur et à mesure. Par exemple pour regle(0,15,4) ça donne :

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

2 2 2 2 2 2 2

3 3 3

4

Les valeurs de graduation (1,2,3,4) de chaque nouvelle ligne remplacent celles de la précédente, ce qui donne une seule ligne :

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1

```
void regleIter(int g, int d, int h)
{
int i,pas,j,m;
   for (i=1,pas=1,m=(g+d); i<=h; i++,pas*=2,m/=2)
      for (j=0; j<m;j++)
         trace((pas+pas*j)-1,i);  // -1 pour partir du bord
}
```

b. Tracé de cercle

Avec la récursion il souvent difficile de départager le très simple du très complexe et une description récursive apparemment élémentaire peut faire apparaître des figures très complexes notamment dans le domaine graphique.

Par exemple nous voulons maintenant, sur le mode de la règle graduée non plus tracer des traits (que nous avons remplacer par des chiffres ci-dessus) mais tracer des cercles.

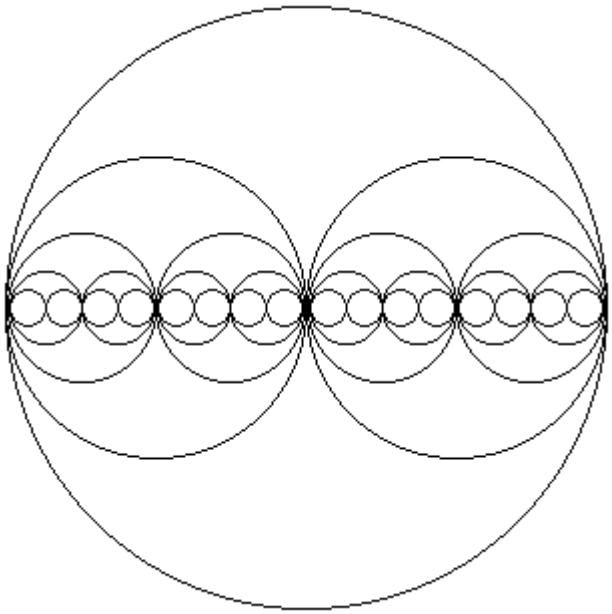
La fonction a en paramètre une position (x,y) et un rayon r, elle trace le cercle puis s'appelle elle-même récursivement deux fois une en (x-r/2,y) et une (x+r/2). Le test d'arrêt c'est r inférieur à une valeur donnée, ça donne :

```
void deuxCercles(int,x, int y, int r)
{
int r2=r/2;

   if (r>5){
          trace_cercle(x,y,r);
          deuxCercle(x-r2,y,r2);
          deuxCercle(x+r2,y,r2);
   }
}
```

Cette fois il vaut mieux passer en mode graphique et utiliser un environnement comme allegro ou SDL qui fournissent des primitives de dessin. Voici l'exécution pour au départ x,y au centre de l'écran et 150 pixels de rayon soit de l'appel :

```
recCercles(SCREEN_W/2,SCREEN_H/2,150);
```



05RI01

Voici la fonction complète :

```
void recCercles(int x, int y, int r)
{
int r2=r/2;
int color;
   if (r>5){
      color=makecol(rand()%256,rand()%128, rand()%64);
      circlefill(screen,x,y,r,color);
      recCercles(x-r2, y, r2);
      recCercles(x+r2, y, r2);
   }
}
```

La fonction makecol() produit une couleur aléatoire et la fonction circlefill() trace un cercle (se référer à la documentation allegro ou SDL pour tester)

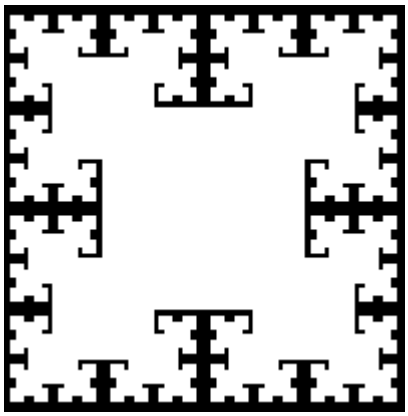
c. Tracé de carrés

Même principe mais cette fois avec des carrés. Au départ on trace un carré à partir d'une position x,y et d'une moitié de côté. La fonction a trois paramètres : la position x,y,et le demi côté c. Elle s'appelle elle-même quatre fois une pour chaque coin du carré avec un côté divisé par 2 à chaque fois. La récursion a lieu jusqu'à ce que le côté soit au-dessous d'une valeur seuil :

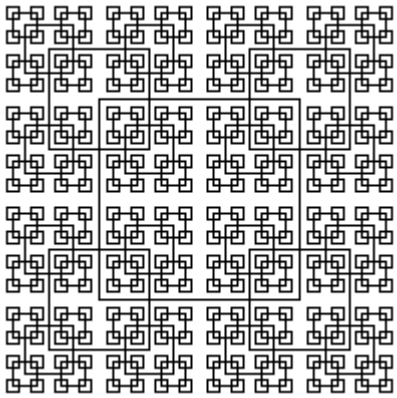
```
void carre(int x,int y,int c)
{
int cc=c/2
   if (c>2){
       // ici dessiner le carré de centre x,y et de demi côté c
       carre(x-c,y-c,cc);
       carre(x+c,y-c,cc);
```

```
        carre(x+c,y+c,cc);
        carre(x-c,y+c,cc);
    }
}
```

On peut obtenir différentes figures, selon le choix des couleurs et un tracé plein ou juste le tour des carrés. Le premier exemple ci-dessous est obtenu avec des carrés pleins en blancs sur fond noir, le second avec juste le tour de carrés noir sur fond blanc :



05RI02



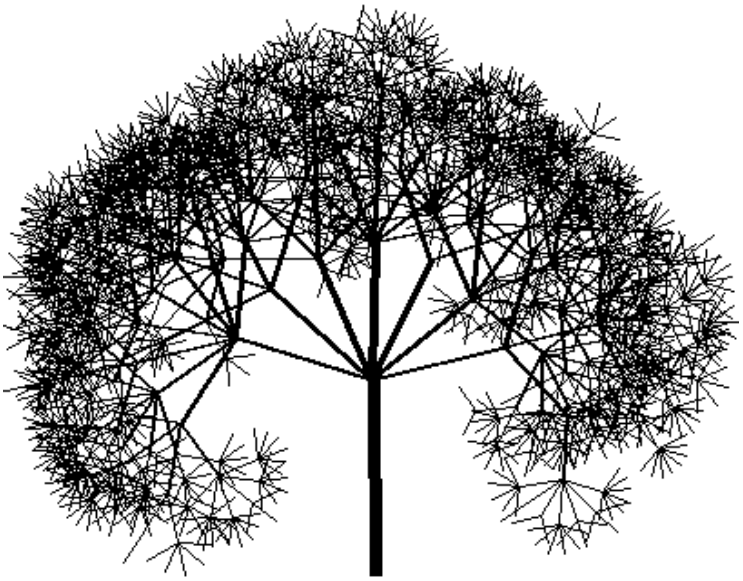
05RI03

La fonction ci-dessous utilise la fonction rect qui trace le contour d'un rectangle de l'environnement allegro.

```
void carre(int x, int y, int c)
{
int cc=c/2;
int color;
   if (c>2){
      color=makecol(rand()%256,rand()%128, rand()%64);
      rect(screen,x-c,y-c,x+c,y+c,color);
      carre(x-c,y-c,cc);
      carre(x+c,y-c,cc);
      carre(x+c,y+c,cc);
      carre(x-c,y+c,cc);
   }
}
```

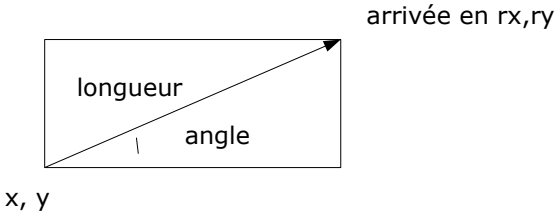
d. Tracé d'un arbre

L'objectif est d'obtenir quelque chose comme le dessin ci-dessous :



05RI04

Pour ce faire on trace un segment et récursivement, à l'issue de chaque segment on trace n segments nouveaux. Chaque segment est tracé à partir d'une position x,y, d'une longueur et d'un angle donné ce qui détermine une position rx, ry d'arrivée du segment :



La position d'arrivée du segment, la fin d'une branche de l'arbre sera la position de départ de plusieurs autres branches. Il est donc important de récupérer cette position d'arrivée du segment pour pouvoir tracer les suivants. Pour une position x,y, les coordonnées du point rx,ry sont obtenues en appliquant les formules mathématiques :

```
rx =  x + longueur * cosinus de l'angle
ry =  y + longueur * sinus de l'angle
```

Voici la fonction de tracer de segment de base, sans couleur et qui suppose une fonction de tracer de trait fournie par un environnement de développement graphique :

```
#define PI        3.1416
#define DEGRE     2*PI/360.0
void segment(int l,int x,int y,int angle, int*rx,int*ry)
{
int i;
   // récupération des coordonnées du point d'arrivée
   *rx = x + l*cos(angle*DEGRE);
   *ry = y – l*sin(angle*DEGRE);
```

```
   // plusieurs traits pour faire l'épaisseur des branches en
   // fonction de la longueur
   for (i=0; i<l/15; i++)
      tracer_ligne(x+i,y,*rx+i,*ry);
}
```

La position d'arrivée du segment est transmise au contexte 'appel vie un passage par référence aux paramètres pointeurs int*rx,*ry.

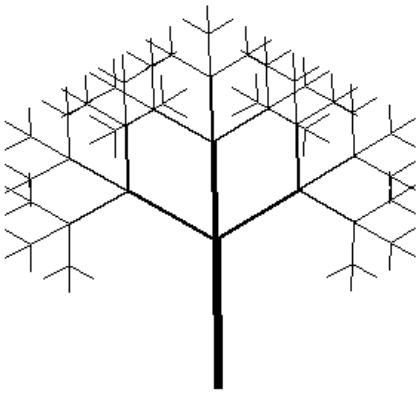
Voici la même fonction adaptée pour l'environnement allegro de programmation graphique :

```
void segment(int l,int x,int y,int angle, int*rx,int*ry)
{
int i,color;
   *rx = x + l*cos(angle*DEGRE);
   *ry = y - l*sin(angle*DEGRE);
   color=makecol(rand()%32,rand()%256,rand()%128);
   //color=makecol(0,0,0);  // pour avoir du noir
   for (i=0; i<l/15; i++)
      line(screen,x+i,y,*rx+i,*ry,color);
}
```

Une fois le tracer de segment résolu, reste l'essentiel : tracer l'arbre. En gros cette nouvelle fonction dessine_arbre() trace un segment d'une hauteur donnée à partir d'une position x,y et selon un angle donné. La position d'arrivée du segment tracé est récupérée, la hauteur est diminuée et si la hauteur est toujours supérieure à 1 pixel on appelle récursivement la fonction dessine_arbre() n fois. Si n est fixe le nombre de branches sera toujours le même mais on peut introduire une valeur aléatoire pour qu'il n'y ait pas toujours le même nombre de branches. Reste la question de l'angle pour le tracé de chaque nouveau segment à chaque noeud de l'arbre. L'idée est de répartir les directions entre -π/2 et π/2 soit un éventail de 180 degrés divisé selon le nombre des branches à tracer et incliné à chaque fois de 90 degrés. Pour la fonction de base suivante la hauteur h reste fixe et à chaque passage elle est diminuée d'un tiers. Le nombre des branches est fixe également, à chaque noeud il y a toujours le même nombre de branche, ce qui donne :

```
void dessine_arbre(int h, int x,int y, int angle)
{
int rx,ry,n,i,ev;

   segment(h,x,y,angle,&rx,&ry);
   h=h*2/3;
   if (h>1){
      n=3;
      ev=180/n;
      for (i=1; i<=n; i++)
         dessine_arbre(h,rx,ry,angle-90-ev/2+i*ev);
   }
}
```



05RI05

Dans la version ci-dessous des déséquilibres sont ajoutés qui introduisent des dissymétries, c'est le paramétrage qui donne la première image de l'arbre ci-dessus :

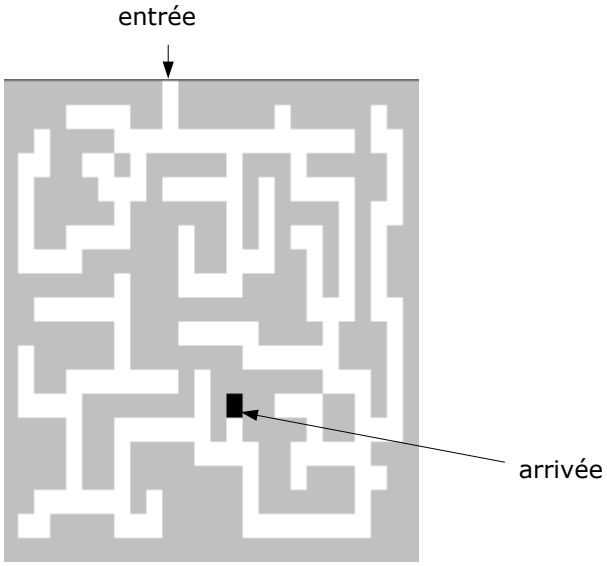
```
void dessine_arbre(int h, int x,int y, int angle)
{
int rx,ry,n,i,ev;
   // disymétrie aléatoire sur la taille des segments
   // (entre 0 et 10% en plus de h)
   h= h+0.1*(rand()%h);

   segment(h,x,y,angle,&rx,&ry);
   h=h*2/3;
   if (h>1){
                  // le nombre de branches est compris entre 1 et 7
      n=1+rand()%8;
      ev=180/n;
      for (i=1; i<=n; i++)
         dessine_arbre(h,rx,ry,angle-90-ev/2+i*ev);
   }
}
```

3. Créations et Jeux

a. Trouver un chemin dans un labyrinthe

Le labyrinthe est situé dans un rectangle avec une ouverture sur un bord au départ. Une case du labyrinthe correspond à un trésor à atteindre. Il n'y a pas de circuit bouclé dans le labyrinthe.



05RI06

Notre objectif est de faire un programme capable de trouver le butin en cherchant le chemin à travers un labyrinthe. Nous allons travailler en mode console pour faire un test.

Le labyrinthe

Le rectangle dans lequel se trouve le labyrinthe est une matrice d'entier de TX par TY déclaré en global et initialisée avec le dessin d'un labyrinthe dans le programme :

```
#define TX           26
#define TY           20

static int MAT[TY[TX]={

   {0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
   {0,0,0,0,1,1,1,1,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0},
   {0,0,1,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,0},
   {0,1,1,0,0,1,1,0,1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,1,0},
   {0,1,0,0,0,0,1,1,1,0,1,1,1,1,1,0,1,0,1,1,1,1,0,0,1,0},
   {0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,1,0,0,0,0,1,0,1,1,0},
   {0,1,0,0,1,1,1,1,1,0,0,0,1,0,0,1,0,1,0,1,1,0,1,0,1,0,0},
   {0,1,1,1,1,1,0,0,0,0,0,0,1,0,0,1,1,1,0,0,1,0,1,0,1,0,0},
   {0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,0,0,0,0,1,0,1,0,1,0,0},
   {0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,1,1,0},
   {0,0,0,0,0,0,0,0,1,0,0,0,1,1,1,1,1,0,0,0,0,1,0,0,0,1,0},
   {0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,1,0},
   {0,1,0,0,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,0,1,1,1,0,1,0},
   {0,1,1,1,1,0,0,0,0,0,0,0,1,0,2,0,0,1,1,1,0,0,1,0,1,0},
   {0,0,0,0,1,0,0,1,1,1,1,1,1,0,1,0,0,0,0,1,0,0,1,1,1,0},
   {0,0,0,0,1,0,0,1,0,0,0,0,1,1,1,1,0,0,1,1,1,1,1,0,0,0},
   {0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,0},
   {0,0,1,1,1,1,1,1,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0},
   {0,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0},
   {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0} };
```

Dans la matrice les positions à 0 correspondent aux murs, les positions à 1 à un chemin, la position à 4 est le point de départ et la position à 2 le point d'arrivée. Ces nombres vont permettre de déterminer des couleurs pour l'affichage mais aussi le trajet à effectuer. Le but à atteindre et les murs sont identifiés par deux macros :

```
#define BUT          2     // pour la couleur du trésor
#define MUR          0     // pour la couleur mur
```

La matrice du labyrinthe est toujours utilisée comme variable globale dans la suite du programme et ne fait pas l'objet d'un paramètre de fonction.

Affichage du labyrinthe

Pour afficher ce labyrinthe en mode console, nous utilisons trois fonctions fournies dans l'environnement : gotoxy() pour déplacer le curseur en écriture à une position x,y de la fenêtre console textcolor() pour obtenir une couleur parmi les 16 possibles et putchar() pour afficher un caractère à la position du curseur dans la fenêtre console. gotoxy() et textcolor() sont données en annexe et putchar() est dans la librairie standard stdio.h.

Le premier point est d'afficher un caractère à une position donnée et d'une couleur donnée. Le caractère affiché est toujours un espace, ce caractère prend toujours la couleur du fond (sans lettre par dessus). En mode console la couleur est codée sur 4 bits, sur les 4 premiers bits de l'octet tient la couleur de la lettre et sur les 4 suivants la couleur du fond. Pour avoir une couleur de fond entre 0 et 15 il faut décaler une valeur choisie entre 0 et 15 de 4 octets vers la gauche. Voici notre fonction d'affichage d'une position :

```
void affiche_posi(int x, int y, int color)
{
   gotoxy(x,y);
   textcolor( color<<4);
   putchar(' ');
}
```

Ensuite il reste à appeler cette fonction pour chaque position de la matrice afin d'obtenir le dessin dans la fenêtre console. Chaque position de la matrice contient sa couleur : 0 pour noir, 1 pour bleu foncé, 2 pour vert foncé, 4 pour rouge foncé. Voici la fonction d'affichage du labyrinthe :

```
void affiche_laby()
{
int x,y,color;
   for (y=0; y<TY; y++)
      for(x=0; x<TX; x++)
         affiche_posi(x,y,MAT[y][x]);
}
```

La recherche du chemin

Pour voir l'affichage se faire au fur et à mesure nous allons avoir besoin de le ralentir, pour ce faire voici une petite fonction d'attente selon la durée passée en paramètre (en milliseconde mais ça dépend des environnements) :

```
void attendre(int dure)
{
int start=clock();
   while (clock()<start+dure){}
```

```
}
```

Les directions possibles pour la recherche sont 0 vers nord, 1 vers est, 2 vers sud et 3 vers ouest. L'objectif est de trouvé la case sur BUT, de colorer la recherche effectuée puis le chemin trouvé, ce qui donne en mode console la fonction récursive suivante :

```
int chemin (int x,int y,int olddir)
{
int i,nx,ny;

   // 1
   if (MAT[y][x]==BUT)
      return 1;
   // 2
   if (MAT[y][x]!=MUR ){
      // 3
      attendre(40);
      affiche_posi(x,y,BLEUCLAIR);
      // 4
      for (i=0;i<4;i++)
         if(i!= (olddir+2)%4 ){
            // 5
            nx=x+(i%2)*(2-i);
            ny=y+((i+1)%2)*(i-1);
           // 6
            if (chemin(nx,ny,i)){
               // 7
               attendre(45);
               affiche_posi(x,y,ROUGE);
               return 1;
            }
         }
   }
   return 0;
}
```

(1) C'est le test d'arrêt de la fonction récursive lorsque le but est atteint : elle retourne 1

(2) Sinon, il faut que la position x,y soit sur un chemin, si on est sur un mur pas d'appel récursif non plus et retour de 0

(3) Si on est sur un chemin, affichage de la position en bleu clair. La fonction attendre provoque un ralentissement pour rendre l'animation de la recherche visible (sinon c'est quasi instantané)

(4) Ensuite partir dans toutes les 4 directions pour continuer la recherche. Il faut éviter cependant de retourner en arrière, c'est à dire qu'il faut éviter la direction inverse de celle d'où l'on arrive (si on allait vers le sud, il faut éviter juste après d'aller vers le nord). On remarque aussi l'importance de l'ordre donné pour la recherche. Ici on essaie toujours dans le même ordre nord-est-sud-ouest. Le simple fait d'inverser cet ordre et faire ouest-sud-est-nord donne une recherche très différente. L'idéal est d'avoir toujours un ordre différent, c'est à dire une combinaison aléatoire des chiffres 0,1,2,3 à chaque tour. Ce point est abordé un peu plus loin avec la création du labyrinthe.

(5) Là il s'agit de prendre les nouvelles coordonnées nx,ny en fonction de la direction i. Nous aurions pu faire un switch et procéder au cas par cas mais, pour une direction i donnée, la formule (i%2)*(2-i) donne -1, 0 ou 1 pour à ajouter à x et la formule ((i+1)%2)*(i-1) donne -1,0 ou 1 pour ajouter à y.

(6) Appelle récursif de la fonction chemin() pour la nouvelle position nx,ny en précisant la direction prise (ce qui permet en 4 d'éviter le retour en arrière). La fonction retourne 0 si rien trouvé et 1 si le but est atteint.

(7) Si le but est atteint, la position trouvée passe en rouge et la fonction retourne 1 également de sorte que le chemin qui a mené à la solution passe progressivement en rouge depuis le but jusqu'au point de départ, au fur et à mesure du dépilement des fonctions concernées.

b. Création d'un labyrinthe

Le principe pour la création d'un labyrinthe sans circuit bouclé à l'intérieur est de creuser au départ un bloc de mur, c'est à dire une matrice de 0 (MUR). On creuse un chemin de façon à ne jamais recouper le chemin déjà creusé. Pour ce faire on avance dans la galerie en creusant le mur de deux en deux cases dans une des quatre directions possibles choisie au hasard. A chaque fois les cases creusées prennent la valeur 1 définie par la macro CHEMIN. Seule les cases sur MUR sont creusables ce qui fait qu'il y a impossibilité de revenir en arrière ou de recouper le chemin déjà creusé.

Avant d'aborder la fonction de création du labyrinthe voici t sont obtenues des suites de directions toujours différentes

Suite toujours différente de directions

Chaque suite de directions (0 pour nord, 1 pour est, 2 pour sud, 3 pour ouest) est stockée dans un tableau de quatre entiers. Au départ elles sont dans l'ordre de 0 à 3 ensuite elles sont mélangées un peu comme des cartes, ce qui donne :

```
void init_dir(int d[4])
{
int i,i1,i2,tmp;
   for (i=0; i<4; i++)
      d[i]=i;

   for (i=0; i<50;i++){
      i1=rand()%4;
      i2=rand()%4;
      if(i1!=i2){
         tmp=d[i1];
         d[i1]=d[i2];
         d[i2]=tmp;
      }
   }
}
```

Le labyrinthe

La fonction a en paramètre la position x,y courante où l'on est dans le labyrinthe. Cette position est sensée pouvoir faire partie du chemin creusé, on commence par la convertir en CHEMIN. A partir de là une suite de directions est stockée dans un

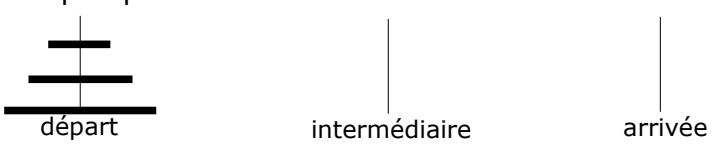
tableau de quatre entiers avec la fonction init_dir() ensuite, dans une boucle for, un switch() est réalisé sur chacune des positions du tableau. Pour chaque direction on va regarder à 2 unités plus loin (+ ou -2 selon direction et en x ou en y) si c'est du MUR et en vérifiant préalablement que l'on reste toujours dans la matrice. Lorsque ces deux conditions sont vérifiées le MUR est remplacé par du CHEMIN sur la case adjacente dans la direction donnée et la fonction est appelée récursivement à partir de la position suivante obtenue (+ ou -2 selon direction en en x ou en y), ce qui donne :

```
void creuse_laby(int x,int y)
{
int d[4];
int i;

   MAT[y][x]=CHEMIN;
   init_dir(d);
   for (i=0; i<4; i++)
      switch(d[i]){
         case 0 :
            if (y>1 && MAT[y-2][x]== MUR){
                MAT[y-1][x]= CHEMIN;
                creuse_laby(x,y-2);
            }
            break;
         case 1 :
            if (x<TX-2 && MAT[y][x+2]== MUR){
                MAT[y][x+1]= CHEMIN;
                creuse_laby(x+2,y);
            }
            break;
         case 2 :
            if (y<TY-2 && MAT[y+2][x]== MUR){
                MAT[y+1][x]= CHEMIN;
                creuse_laby(x,y+2);
            }
            break;
         case 3 :
            if (x>1 && MAT[y][x-2]== MUR){
                MAT[y][x-1]= CHEMIN;
                creuse_laby(x-2,y);
            }
            break;
      }
}
```

4. Les tours de Hanoï

C'est un petit jeu de réflexion, cas d'école typique de la récursivité. On a trois piquets avec au départ des disques de rayons décroissants empilés sur le premier piquet. L'objectif est d'obtenir le même empilement sur le troisième piquet en déplaçant un à un chaque disque et de façon à ce que jamais un plus grand ne soit au-dessus d'un plus petit. Au début la situation est la suivante :



et il s'agit d'arriver à :

En utilisant le piquet intermédiaire alternativement avec les deux autres pour que jamais un plus grand disque ne soit au-dessus d'un plus petit.

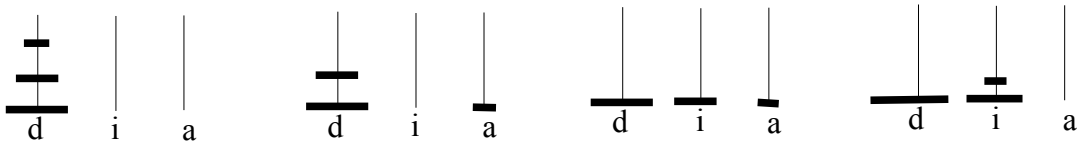
Le jeu peut être envisagé avec n disques et toujours trois piquets. Il s'agit d'écrire le programme qui indique les déplacements successifs qu'il faut effectuer... l'algorithme des tours de Hanoï est assez fascinant. C'est une fonction récursive qui a quatre paramètres : le nombre n des disques, le piquet d de départ, le piquet i intermédiaire et le piquet a arrivée. Le déplacement peut être visualisé avec la fonction suivante :

```
void deplacement(int n, int d, int i, int a)
{
   if (n>0){
      depacement(n-1, d,a,i);    // d vers i avec a intermédiaire

      // d vers a avec i intermédiaire
      printf("aller de %d a %d\n",d,a);

      deplacement(n-1,i,d,a);   // i vers a avec d intermédiaire
         }
}
```

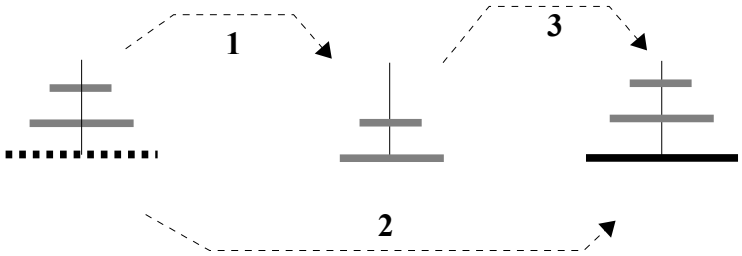
Le principe est le suivant :



A ce stade,

- deux disques sont déplacés de d vers i en passant par a comme intermédiaire
- reste à déplacer le dernier disque en d vers a (avec i comme intermédiaire)
- puis déplacer les deux disque en i vers a en passant par d comme intermédiaire

En résumé nous avons :



1 : appels récursifs de d vers i avec a comme intermédiaire

2 : si un seul disque, alors appels de d vers a avec i comme intermédiaire (fin première pile d'appels récursifs)

3 : appels récursifs de i vers a avec d comme intermédiaire

La fonction peut s'écrire également (en prenant les lettres 'D', 'I', 'A' plutôt que les nombres 1,2,3 pour les poteaux) :

```
void hanoi(int n,int d, int i, int a)
{
   if (n==1)
       printf("aller de %c en %c\n",d,a);
       else{
           hanoi(n-1,d, a, i);
           hanoi(1,d, i, a);
           hanoi(n-1,i, d, i);
   }
}
```

exemple d'appel dans le main() pour n=3 :

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   hanoi(3, 'D', 'I', 'A');
   return 0;
}
```

Remarque :

Avec les nombres 0, 1, 2 comme numéro des piquets d, i, a, nous pouvons à tout moment connaître le numéro du piquet intermédiaire par la formule i = 3 – d – a.

Les trois mouvements peuvent alors s'écrire :

1 : d vers i,	appels récursifs jusque n==1	avec i = 3 – d – a
2 : d vers a ,	pas d'appel récursif n == 1	
3 : i vers a,	appels récursifs jusque n==1	avec i = 3 – d – a

Cette formule est nécessaire pour tracer un arbre d'appel de la fonction.

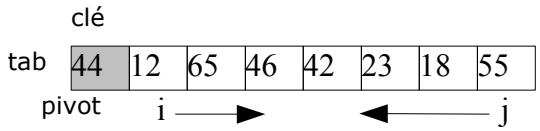
5. 4. Tableaux et Matrices

a. Tri rapide d'un tableau de nombres

Les algorithmes de tris ont constitué un espace de travail et de recherche de l'informatique qui a fait l'objet de nombreuses découvertes à ces débuts. Ces algorithmes peuvent être utiles en programmation système et diverses autres circonstances "low level", à chaque fois qu'il est nécessaire d'ordonner et d'organiser des données. Ce sont maintenant autant de classiques bien intéressants à étudier quoiqu'il soit plus rare d'avoir à les implémenter. Afin d'illustrer une utilisation "diviser pour résoudre" de la récursivité nous allons étudier l'algorithme dit de "tri rapide" (quicksort) appliqué à un tableau de nombres.

Soit un tableau d'entiers rempli de valeurs, l'objectif est de les ordonner en ordre croissant. Le principe du tri rapide est de partitionner récursivement le tableau et d'organiser le tri au fur et à mesure sur chacune des parties obtenues.

Au départ on prend une valeur clé qui est le bord gauche de la partition courante. L'indice de cette valeur clé est le pivot à partir duquel la partition a été effectuée. Ensuite on parcourt cette partition en remontant simultanément de la gauche vers la droite et en descendant de la droite vers la gauche. Le processus est alternatif : on avance d'une nouvelle position i vers la droite puis on avance d'une nouvelle position j vers la gauche :



● l'avance de i et j se passe de la façon suivante :
i progresse en cherchant une valeur > cle : tant que tab[i]<=cle faire i=i+1 fin
j progresse en cherchant une valeur <= clé : tant que tab[j]>cle faire j=j-1 fin

● une fois i et j obtenus, si la borne gauche i reste inférieure à la borne droite j alors échanger les valeurs tab[i] et tab[j] :

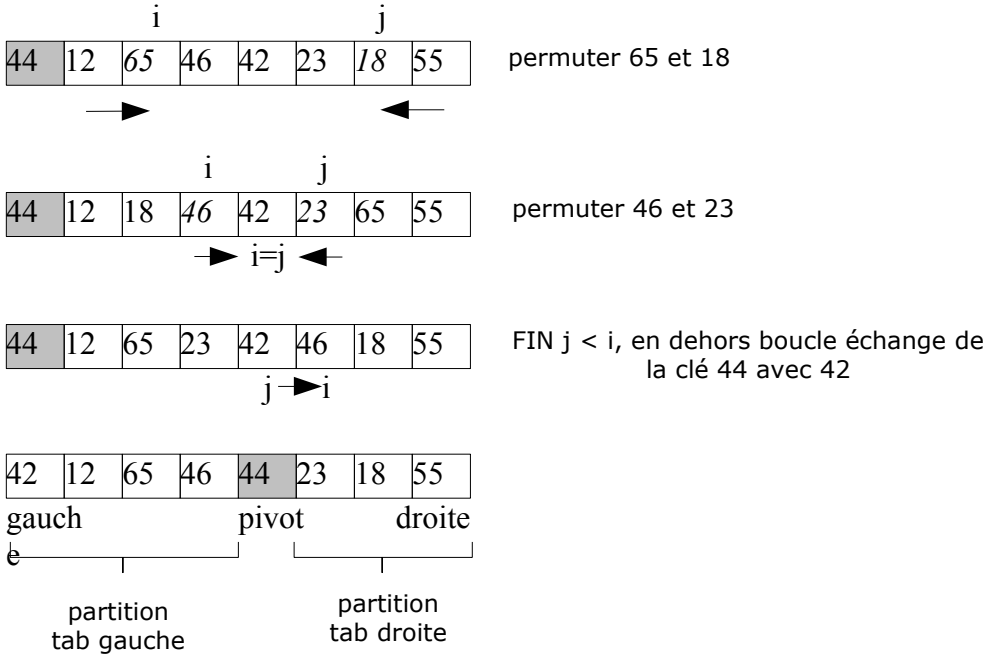
SI i < j ALORS permuter(tab, i, j) et faire i=i+1, j=j-1

● recommencer tant que i<=j

● à l'issue de la boucle échanger tab[pivot] avec tab[j], c'est à dire la valeur de clé vient en tab[j] et j va servir à définir le nouveau pivot pour la partition suivante

● retourner j

Le processus peut se visualiser ainsi :



A la fin, la valeur clé de partition tab[pivot] est toujours positionnée de façon à ce que :

tab [gauche à pivot-1] <= clé < tab [pivot+1 ….à droite]

La tableau tab est alors partitionné en deux sous tableaux un de gauche à pivot-1 et un de pivot +1 à droite. La récursion s'effectue sur ces sous tableau tant que gauche < droite. Voici l'algorithme final décomposé en trois fonctions, partitionner, permuter, trier avec récursion :

```
int partitionner(int t[], int gauche,int droite)
{
int cle, i, j;
   cle=t[gauche];
   i=gauche+1;
   j=droite;
   while(i<=j){
      while(t[i]<=cle)  i++;
      while(t[j]>cle)   j--;
      if(i<j)
         permuter(t,i++,j--);
   }
   if(t[j]<t[gauche])
      permuter(t,gauche,j);
   return j;
}

void permuter(int t[], int i, int j)
{
int tmp;
   tmp=t[i]; t[i]=t[j]; t[j]=tmp;
}

void trier(int t[], int gauche, int droite)
{
int pivot;
   if (gauche<droite){
      pivot=partitionner(t,gauche,droite);
      trier(t,gauche,pivot-1);
      trier(t,pivot+1,droite);
   }
}
```

C. Mise en pratique récursivité

Exercice 1 :
Afficher avec une fonction récursive tous les nombres entre n et n' entrés par l'utilisateur

Exercice 2
Qu'impriment les programmes suivants ?

```
// programme 1
void f(int n)
{
   if (n>0){
       f (n-3);
       printf("%3d\n",n);
```

```
        f (n-2);
    }
}

int main()
{
    f(6);
    return 0;
}

// programme 2
void f(int n)
{
    if (n>0){
        f (n/10-100);
        printf("%3d\n",n);
        f (n/5-200);
    }
}
int main()
{
    f (10000);
    return 0;
}
```

Exercice 3

Remplacer les fonctions f(), g(), h() ci-après par des variantes itératives équivalentes (plus ou moins). Comparer dans chaque cas la quantité de mémoire utilisée et le temps de calcul nécessaire pour l'une et l'autre des variantes.

```
void f(void)
{
    if (getchar() == ' ')
        f ();
}

void g(int n)
{
    int i;
    if (n>0){
        scanf("%d",&i);
        g(n-1);
        printf("%d\n",i);
    }
}

int h(int n)
{
    return n<0 ? 0 : (n==0 ? 1 : h(n-1) + h(n-2));
}
```

Exercice 4

Écrire une fonction prenant un argument entier et renvoyant la somme des chiffres décimaux constituant l'argument. Comparer deux variantes de la solution à ce problème, l'une récursive et l'autre itérative.

Exercice 5

Ecrire un programme destiné à lire une succession de nombres réels en virgule flottante et afficher la somme des 1e, 3e, 6e, 10e, 15e, (etc.) éléments de cette

suite. Tout caractère non numérique sera interprété comme le signal d'arrêt de cette suite de nombres.

Exercice 6
Ecrire un programme destiné à lire un chiffre décimal d. Afficher systématiquement chacun des entiers positifs x inférieurs à 100 qui comportent d pour x et pour x^2

Exercice 7 :
Faire une fonction récursive qui à partir de n entré par l'utilisateur calcule la somme n+n-1+n-2... jusque n=0.

Exercice 8 :
Faire une fonction récursive qui à partir de n entré par l'utilisateur calcule la somme $u_n = 1 + 2^4 + 3^4 + 4^4 + \ldots + n^4$

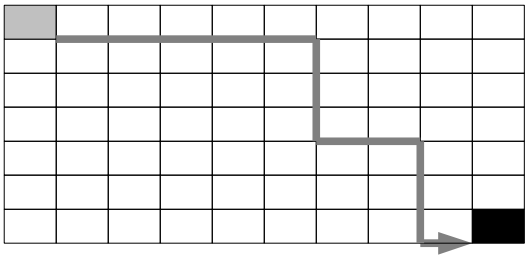
Exercice 9 :
Sur le modèle de la suite de Fibonacci faire une fonction récursive pour calculer la suite à partir de n entré par l'utilisateur : f(n) = n-4 + n-3 + n-2 + n-1

Exercice 10 :
Sur le modèle du calcul factoriel faire une fonction récursive qui calcule la suite pour n entré par l'utilisateur : f(n) = 2*n*n-1 … pour n>0

Exercice 11 : strlen en récursif
Faire une version récursive de la fonction strlen() qui retourne la longueur d'une chaine de caractères donnée en paramètre.

Exercice 12 : détecteur de palindrome
Un palindrome est un mot ou une phrase qui se lit aussi bien à l'envers qu'à l'endroit (sans tenir compte des espaces), par exemple "radar", "kayak", "abccba" ou une phrase "esope reste et se repose". La phrase ou le mot sont entrés par l'utilisateur et une fonction récursive retourne si oui ou non la chaine de caractères donnée en paramètre est un palindrome.

Exercice 13 : Parcours systématique d'une matrice
Soit une matrice MAT d'entiers de TX par TX faire le programme qui affiche tous les chemins directes possibles qui partent de MAT[0][0] pour arriver en MAT[TY-1][TX-1] sans jamais retourner en arrière ou faire des circuits. C'est à dire sur le schéma, tous les trajets qui peuvent aller directement en avançant ou en descendant de la case grise en haut à gauche à celle en bas à droite :

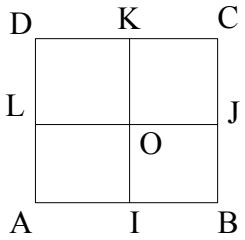


L'affichage est soit graphique avec allegro ou conio, soit en en texte le trajet étant une suite de positions : (0,0)-(0,1)-(0,2)-(...,...)-(TX-1, TY-1)

Exercice 14 : Parcours aléatoire d'une matrice

Soit une matrice de TX par TY faire une fonction récursive de remplissage aléatoire. Chaque position ne peut être visitée qu'une fois et sert de départ pour visiter d'autres positions.
Faire un affichage graphique avec conio ou sous allegro.

Exercice 15 : Dégradé de couleur dans un carré
On suppose qu'on a une certaine palette de couleurs style arc en ciel avec un dégradé de couleurs. On donne une couleur au hasard à chaque coin A, B, C, D d'un carré tracé à l'écran. Puis on donne la moyenne de ces quatre couleurs au centre du carré O.



On calcule également les couleurs des points I, J, K, L par moyenne des deux couleurs voisines. Puis on recommence avec les quatre carrés obtenus, et ainsi de suite jusqu'à ce que les carrés aient des côtés de l'ordre du pixel.

Faire un programme de test de préférence en mode graphique sous allegro.

Exercice 16 : Tracé de carrés
Au départ on a une position (x,y), par exemple le centre de l'écran et une taille de segment s. Faire deux fonctions qui tracent récursivement 8 carrés répartis de façon fractale les uns autour des autres. La première fonction trace ses carrés à partir d'une taille maximum donnée en entrée et vers l'intérieur. La seconde fonction trace des carrés de façon extensive, vers l'extérieur jusqu'à atteindre une taille maximum.

Exercice 17 : tracer une spirale
Faire une fonction récursive qui permet de tracer une spirale rectangulaire. Au départ nous avons une longueur L pour le premier segment et la taille maximum qu'un segment peut atteindre.

Exercice 18 : Placer des fruits dans l'arbre
Modifier la fonction récursive dessine_arbre() étudiée dans le cours de façon à dessiner le segment terminal en vert et à insérer de façon aléatoire des fruits de couleur dans l'arbre.

Exercice 19 : faire un labyrinthe avec croisements de chemin
Modifier la fonction de création de labyrinthe donnée dans le cours afin de produire des labyrinthes avec croisements de chemin. Comment faire une fonction de recherche pour un tel labyrinthe ?

Exercice 20 : Base du démineur
Le jeu consiste à retirer des mines cachées et disposées sur un terrain. Le terrain est une grille. Le joueur clic sur une case s'il y a une mine il a perdu. S'il y a une mine ou plusieurs mines adjacentes, le nombre de ces mines est indiqué dans la case cliquée. S'il n'y a pas de mine le programme découvre automatiquement toute la zone ceinturée de cases adjacentes à des mines. Chacune de ces cases indique

le nombre des cases minées voisine. En se servant de ces indications le joueur doit déduire où sont les mines. Il a la possibilité de déposer des drapeaux pour neutraliser une case...

- □ Notre objectif est de mettre en place une base pour le jeu :
- □ le terrain est une matrice de nombres. Les case non minées sont à 0 ou indiquent combien il y a de mines à proximité les autres à une valeur MINE
- □ faire une fonction qui initialise le terrain avec des mines ainsi que toutes les positions adjacentes
- □ faire une fonction de recherche appelée lorsque le joueur clic sur le terrain (dans une case de la matrice sous-jacente.

Exercice 21 : récursivité et pile, visualiser le déplacement des anneaux des tours de hanoï

L'objectif est d'écrire un programme pour visualiser l'algorithme des tours de Hanoï vu en cours et pour un nombre n d'anneaux avec n >= 3. La récursivité devra être éliminée via l'utilisation de piles, une pile par poteau. Les anneaux sont représentés par des entiers qui donnent la taille de l'anneau.

Avant de commencer :

- □ Pourquoi un poteau peut-il se représentée par une pile ?
- □ Dessinez l'arbre d'appels de la procédure Hanoi donnée en cours pour n=3.
- □ Avec votre arbre d'appels, donnez l'ordre de déplacement de chaque anneau tel que visualisé dans le printf de la procédure Hanoi

Ensuite :

Reprendre la procédure Hanoi vue en cours mais en représentant chaque tour par une pile. Le déplacement visuel du printf sera remplacé par un déplacement effectif d'un anneau entre deux piles (tours).

1) Ecrire une fonction qui affiche le contenu d'une pile d'anneaux (1 anneau = 1 entier).

2) Déclarer trois piles qui représentent les trois poteaux du jeu, et écrire une fonction qui initialise le jeu en empilant les n anneaux correctement sur l'un des trois poteaux. Ecrire également une fonction qui affiche à l'écran l'état du jeu à un instant donné.

3) Ecrire une fonction qui déplace les n anneaux sur l'un des deux poteaux libres en respectant les règles du jeu. Faire en sorte que l'affichage à l'écran montre bien le déplacement de chaque anneau.