Récursivité

L2 Informatique – Université Paul Sabatier

1 Introduction

Un problème est calculable s'il existe une fonction calculable par machine de Türing qui résout ce problème. Définir la fonction à partir de fonctions de base paraît simple. $Somme(x, y)$ peut se définir par $x + y$ en utilisant l'opération $+$ comme opération de base. Ceci peut paraître plus complexe quand le nombre d'utilisations de l'opération de base est variable.

Supposons que l'opération de multiplication $*$ est une opération de base. La fonction $N! = 1 * 2 * ... * N$ ne peut pas être obtenue par une composition explicite et simple de fonctions de base. Pour ce faire, nous utilisons la notion de récursion:

$$N! = \begin{cases} 1 & \text{si } N = 0 \\ N * (N-1)! & \text{si } N \in \mathbb{N}^* \end{cases}$$

C'est une définition effective car $N!$ est défini en terme de $(N-1)!$ et d'opérations que l'on sait calculables.

La notion de fonction récursive est introduite de deux manières différentes que l'on peut montrer équivalentes:

- Une fonction récursive est une fonction calculable à plusieurs arguments sur les entiers naturels, donc une fonction de $\bigcup_{k\geq 0} \mathbb{N}^k \to \mathbb{N}$.
- Une fonction récursive est définie par composition de fonctions de base et en exploitant des mécanismes de récursion.

Nous détaillons la deuxième approche puis nous étudions différents aspects de la programmation récursive qui seront illustrés en langage C.

2 Fonctions récursives

Pour donner un sens précis à la notion de fonction récursive nous décrivons les mécanismes de la construction récursive et nous énoncerons les résultats sur leur calculabilité.

- En première étape nous donnons la définition des fonctions "Primitives Récursives **FPR**". Bien qu'il soit possible d'exprimer la plupart des algorithmes par des FPR, il a été démontré qu'il existe des fonctions calculables qui ne sont pas **FPR**.
- Nous complétons, en deuxième étape, la définition selon **FPR** par d'autres fonctions et schémas afin d'atteindre la calculabilité au sens des MT et donc la notion de fonction récursive.

2.1 Fonctions Primitives Récursives (FPR)

Les fonctions primitives récursives forment un sous-ensemble des fonctions de $\mathbb{N}^k \to \mathbb{N}$ pour $k \geq 0$ qui peuvent être définies comme:

- une fonction primitive récursive de base
- la composition de deux **FPR**
- l'application du schéma de récursion primitive sur des **FPR**

2.1.1 Les fonctions primitives récursives de base

1. La fonction 0() qui est sans argument, elle retourne 0
2. La fonction successeur $\sigma \in \mathbb{N} \to \mathbb{N}$ définie par $\sigma(N) = N + 1$
3. Les fonctions de projection: $\mathbf{prj}_i^k \in \mathbb{N}^k \to \mathbb{N}$ définies par $\mathbf{prj}_i^k(N_1, N_2, \ldots, N_k) = N_i$.

2.1.2 La composition

Etant données les fonctions suivantes:

- $g \in \mathbb{N}^l \to \mathbb{N}$ une fonction à l arguments,
- $h_1, h_2, \ldots h_l \in \mathbb{N}^k \to \mathbb{N}$ des fonctions à k arguments,

la composition de g et de $h_1, h_2, \ldots, h_l$, dénotée $g \circ [h_1, \ldots, h_l]$, est la fonction $f : \mathbb{N}^k \to \mathbb{N}$ définie par:

$$f(\bar{N}) = g(h_1(\bar{N}), h_2(\bar{N}), ...h_l(\bar{N}))$$

où $\bar{N}$ représente le k-uplet d'arguments $N_1, \ldots, N_k$.

2.1.3 La récursion primitive: REC

Etant données les fonctions suivantes:

- $g \in \mathbb{N}^k \to \mathbb{N}$ une fonction à k arguments,
- $h \in \mathbb{N}^{k+2} \to \mathbb{N}$ une fontion à $k+2$ arguments,

la fonction $f = \mathbf{REC}(g,h) \in \mathbb{N}^{k+1} \to \mathbb{N}$ à $k+1$ arguments se définit par:

- $f(\bar{N}, 0) = g(\bar{N})$
- $f(\bar{N}, M+1) = h(\bar{N}, M, f(\bar{N}, M))$

2.1.4 Définition de l'ensemble des Fonctions Primitives Récursives

Définition 1 *L'ensemble des* **FPR** *est le plus petit ensemble contenant les fonctions de base et stable par les schémas de composition et de récursion primitive. Autrement dit, les fonctions primitives récursives sont définies par un nombre quelconque d'applications de la composition et de la récursion primitive en partant des* **FPR** *de base.*

Théorème 1 *Les* **FPR** *sont calculables par une procédure effective: il est possible d'associer une machine de Turing qui termine à chacune des* **FPR**.

2.1.5 Exemples

- Les constantes naturelles: $\mathbf{j}() = \underbrace{\sigma(\sigma(\ldots}_{j \text{ fois}}(\mathbf{0}())\ldots)$

  $\mathbf{3}() = \sigma(\sigma(\sigma(\mathbf{0}())))$

- $\mathbf{plus}(n_1, n_2)$ se définit par récursion primitive: $\mathbf{plus} = \mathbf{rec}(\mathbf{prj}_1^1, \sigma \ o \ \mathbf{prj}_3^3)$

  c'est à dire:

  - $\mathbf{plus}(n_1, 0) = \mathbf{prj}_1^1(n_1)$
  - $\mathbf{plus}(n_1, n_2+1) = \sigma(\mathbf{prj}_3^3(n_1, n_2, \mathbf{plus}(n_1, n_2))$

  ou encore:

  - $\mathbf{plus}(n_1, 0) = n_1$
  - $\mathbf{plus}(n_1, n_2+1) = \sigma(\mathbf{plus}(n_1, n_2))$

  Evaluer $\mathbf{plus}(5,3)$

  - $\mathbf{plus}(5,3) = \mathbf{plus}(5, 2+1)$
  - $\mathbf{plus}(5,3) = \sigma(\mathbf{plus}(5,2))$
  - $\mathbf{plus}(5,3) = \sigma(\sigma(\mathbf{plus}(5,1)))$

  - $\mathbf{plus}(5,3) = \sigma(\sigma(\sigma(\mathbf{plus}(5,0))))$
  - $\mathbf{plus}(5,3) = \sigma(\sigma(\sigma(5)))$
  - $\mathbf{plus}(5,3) = 8$

- **prod**(n_1, n_2) se définit par:
  - $\mathbf{prod}(n_1, 0) = 0$
  - $\mathbf{prod}(n_1, n_2 + 1) = \mathbf{plus}(n_1, \mathbf{prod}(n_1, n_2))$

- **puissance**(n_1, n_2) se définit par:
  - $\mathbf{puissance}(n_1, 0) = 1$
  - $\mathbf{puissance}(n_1, n_2 + 1) = \mathbf{prod}(n_1, \mathbf{puissance}(n_1, n_2))$

- **factorielle**(n) se définit par:
  - $\mathbf{factorielle}(0) = \sigma(0())$
  - $\mathbf{factorielle}(n + 1) = \mathbf{prod}(\sigma(n), \mathbf{factorielle}(n))$

- **pred**(n) se définit par:
  - $\mathbf{pred}(0) = 0()$
  - $\mathbf{pred}(n + 1) = n$

- **moins**(x, y) se définit par:
  - $\mathbf{moins}(x, 0) = x$
  - $\mathbf{moins}(x, y + 1) = \mathbf{pred}(\mathbf{moins}(x, y))$

2.1.6 Somme et produit

La somme (resp. le produit) bornée par m d'une fonction primitive récursive $g(\bar{N}, I)$, se définit comme la somme (resp. le produit) des $m + 1$ premières valeurs de I

  - $S(\bar{N}, m) = \sum_{I=0}^{m} g(\bar{N}, I)$
    - $S(\bar{N}, 0) = g(\bar{N}, 0)$
    - $S(\bar{N}, m + 1) = \mathbf{plus}(S(\bar{N}, m), g(\bar{N}, m + 1))$
  - $P(\bar{N}, m) = \prod_{I=0}^{m} g(\bar{N}, I)$
    - $P(\bar{N}, 0) = g(\bar{N}, 0)$
    - $P(\bar{N}, m + 1) = \mathbf{prod}(P(\bar{N}, m), g(\bar{N}, m + 1))$

2.2 Fonctions récursives versus fonctions primitives récursives

Existe-t-il des fonctions calculables qui ne sont pas primitives récursives? **OUI**
C'est démontrable, à la Cantor:

- L'ensemble des fonctions primitives récursives est dénombrable: $\mathbf{FPR} = \{f_1, f_2....f_N....\}$
- la fonction $F(N) = f_N(N)+1$ est calculable car f_N est calculable mais $F(N)$ n'appartient pas à **FPR**

A savoir

- Problème indécidable: Il n'éxiste pas d'algorithme qui répond par OUI ou par NON si une fonction récursive est une fonction primitive récursive.
- Exemples: fonction d'Ackermann, fonction de Sedan...

Nous dénotons l'ensemble des fonctions totales calculables, par **FT**.
Nous constatons donc que l'ensemble $\mathbf{FPR} \subset \mathbf{FT}$

2.3 Caractérisation générale des fonctions récursives

Pour ce faire, nous considérons l'ensemble des μ fonctions récursives partielles μ**FRP**. Il s'agit de la plus petite classe de fonctions partielles construites à partir:

- des fonctions de base
- de la composition
- de la récursion primitive
- de l'opérateur $\mu(f)$ avec $f : \mathbb{N}^{k+1} \to \mathbb{N}$ qui, appliqué à $\bar{N}$, retourne lorsqu'il existe le plus petit Z tel que $f(\bar{N}, Z) = 0$. Plus précisemment, il est défini par:
  $\mu(f)(\bar{N}) = Z \Leftrightarrow$
  $f(I, \bar{N})$ est définie pour tout $I \leq Z \wedge$
  $f(Z, \bar{N}) = 0 \wedge$
  $(\forall I : 0 \leq I < Z \to f(Z, \bar{N}) > 0)$

Théorème 2 **Les ensembles FT et μFRP** *sont dénombrables.*

Les μ fonctions récursives étant partielles, elles ne sont pas toutes calculables.
Nous pouvons conclure que; $\mathbf{FPR} \subset \mathbf{FT} \subset \mu\mathbf{FRP}$

Définition 2 *Les fonctions récursives sont les fonctions de* μ**FRP** *totales.*

3 Programmation Récursive

Nous nous intéressons, dans cette section, à la programmation des fonctions en Langage C.

Méthode informelle : Pour coder une fonction récursive, nous nous inspirons de sa définition pour identifier:

- son nom (identificateur accepté par le langage) et les paramètres de la fonction
- les cas triviaux et les traitements associés
- le passage du problème aux sous-problèmes

Exemple 1

$$N! = \begin{cases} 1 & si\ N = 0 \\ N * (N-1)! & si\ N \in \mathbb{N}^* \end{cases}$$

- *Identificateur :factorielle pour remplacer la notation mathématique* !
- *Paramètre:* $N \in \mathbb{N}$
- *Cas trivial :* $N = 0$
- *Traitement du cas trivial:* 1
- *Passage au sous-problème:*$g : \mathbb{N}^* \to \mathbb{N}$ *avec* $g(N) = N - 1$
- *Traitement du cas non trivial (récursif):*$N > 0$ *:* $N * factorielle(g(N))$

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

Comment la fonction factorielle est évaluée?
Nous rappelons que le paramètre n est traité d'une façon automatique en C, c'est à dire qu'à chaque appel de factorielle(valeur) on réserve un emplacement pour n dans la pile où valeur sera empilée.

f=factorielle(4)

*24 ←4*factorielle(3)*
*6 ← 3*factorielle(2)*
*2← 2*factorielle(1)*
*1 ← 1*factorielle(0)*
1 ←

Nous utilisons une pile à travers les opérations suivantes:

```
type pile[element]
/* pour top et depiler, la pile ne doit pas \^etre vide */
creer(): -> pile          /* donne une pile vide */
empiler:pile*element -> pile /* nouveau element en haut de la pile*/
depiler:pile->pile        /* eliminer le haut de la pile*/
vide: pile-> boolean /*vrai si la pile est vide sinon faux*/
top:pile-> element /*donne le haut de la pile*/
```

```
int factorielleNTIter(int n){
  p=creer();
  while (n!=0){
    p=empiler(p,n);
    n=n-1;
  }
  f=1;
  while (!vide(p){
    n=top(p);
    depiler(p);
    f=f*n;
  }
  return f;
}
```

3.1 Récursivité Terminale ou Non Terminale

Terminale si l'appel récursif est la dernière instruction de la fonction, sinon on dit qu'elle est **Non Terminale**.
La fonction factorielle, telle qu'elle est écrite ci-dessus, est Non Terminale car sa dernière instruction est: n*factorielle(n-1)
Nous donnons sa version terminale comme suit:

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

 /* on utilise*/

 f= factorielleT(n, 1)*/
```

Cette nouvelle écriture va nous permettre de traduire systématiquement une fonction récursive Terminale, en boucle sans utilisation de la pile:

```
int factorielleTIter(int n, int f){
   while (n>0){
     f=f*n;
     n=n-1;
   }
   return f;
}

/* on utilise*/

 f= factorielleTIer(n, 1)*/
```

Ecrire en récursivité **Non Terminale** est souvent naturel pour répondre aux problèmes. Existe-t-il un procédé effectif permettant de traduire une fonction récursive **Non Terminale** en fonction récursive **Terminale**?
NON sans utilisation d'une pile non bornée

3.1.1 Appel récursif, avant ou après

Exemple 2 *Ecrire une fonction récursive permettant d'afficher:*

```
*****
****
***
**
*
```

```
void BHaut(int n){
        if (n>0){
                for (int i=0; i<n; ++i)
                        printf("*");
                printf("\n");
                BHaut(n-1);
        }
}
```

Il suffit d'appeler **BHaut(5)** *pour obtenir l'affichage demandé.*

Exemple 3 *Ecrire une fonction récursive permettant d'afficher:*

```
*
**
***
****
*****
```

```
void BBas(int n){
        if (n>0){
                BBas(n-1);
                for (int i=0; i<n; ++i)
                        printf("*");
                printf("\n");

        }
}
```

Il suffit d'appeler **BBas(5)** *pour obtenir l'affichage demandé.*

3.2 Récursivité Multiple

Une fonction récursive peut effectuer plusieurs appels récursifs.

Exemple 4 *Calculer le terme N de la Suite de Fibonacci.*

Il s'agit d'une séquence qui commence par 0,1 et à partir du 3ème élément, chaque élément est la somme de ses deux prédécesseurs immédiats :
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 45,....
Ceci peut être formalisé, pour calculer le terme N de la suite de fibonacci, par la définition suivante :

$$fib(N) = \begin{cases} 0 & si\ N = 0 \\ 1 & si\ N = 1 \\ fib(N-1) + fib(N-2) & si\ N \geq 2 \end{cases}$$

```
int fibrec (int n) {
  if (n==0)
    return 0;
  if (n==1)
    return 1;
  return fibrec(n-2) + fibrec(n-1);
}
```

Quelques questions se posent :

1. Est ce que la fonction fib est correcte ?
   Nous remarquons que la fonction fibrec, correspond exactement à la définition.

2. Combien d'étapes de calcul sont nécessaires pour calculer fib(n) ?
   Nécessite de construire, une fonction NEtape qui calcule le nombre d'appels de fibrec:

   - si $n < 2$ alors NEtape donne au plus 2 (le test de la valeur de n)
   - si $n > 2$, NEtape retourne NEtape(n-2) + NEtape(n-1)
   - Pour $n = 7, NEtape = 38$
   - Pour $N = 200, NEtape > fib(200) > 2^{138}$

3. Peut-on faire mieux ?
   Voir la solution donnée au chapitre 1 – Introduction au langage C

Exemple 5 Tour de Hanoï: *Le problème des tours de Hanoï est un jeu de réflexion imaginé par le mathématicien français Édouard Lucas. Il consiste à déplacer des disques de diamètres différents d'une tour de départ A à une tour d'arrivée B en passant par une tour intermédiaire C, et ceci en un minimum de coups, tout en respectant les deux règles suivantes :*

- *on ne peut pas déplacer plus d'un disque à la fois,*
- *on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.*

Modèle de solution récursif:

1. *déplacer les n-1 premiers disques de A vers C, en utilisant B*
2. *déplacer le plus grand disque de A vers B*
3. *déplacer les n-1 disques de C vers B, en utilisant A.*

Ceci nous amène à effectuer $2^n - 1$ *déplacements*

Exemple avec n=3 et A=1, B=3, C=2

```
#include <stdio.h>
void Hanoi ( char A, char B, char  C, int n){
   if (n !=0){
          Hanoi (A, C, B, n-1) ;
          printf ("%c------->%c|n", A, B) ;
          Hanoi (C, B, A, n-1) ;
   }
}
int main(){
 Hanoi('A', 'B', 'C', 3);
 return 0;
 }
```

```
   A-->B
     A-->C
          B-->C
               A-->B
          C-->A
     C-->B
A-->B
```

Des dérécursifications de cette opération seront traitées en travaux pratiques (avec et sans pile)

3.3 Récursivité Croisée

Il s'agit, par exemple, de définir deux fonctions qui s'appellent mutuellement:

Exemple 6 *Les deux fonctions pair, impair s'appellent mutuellement. L'invocation, au départ, de l'une d'elles, permettra de savoir si un entier naturel est pair ou impair.*

```
int pair(int n){
 if (n==0)
      return 1;
 return impair(n-1);
}
 int impair(int n){
 if (n==0)
      return 0;
 return pair(n-1);
}
```

3.4 Récursivité Imbriquée

Un appel récursif est dit imbriqué s'il est apparaît dans le calcul d'un paramètre d'un appel récursif.

Exemple 7 *La fonction d'Ackerman définie sur* $\mathbb{N} \times \mathbb{N}$*:*

$$A(M,N) \;=\; \begin{cases} N+1 & si\ M=0 \\ A(M-1,1) & si\ M>0, N=0 \\ A(M-1,A(M,N-1)) & si\ M>0, N>0* \end{cases}$$

```
#include <stdio.h>
int A(int m, int n){
        if (m==0)
                return n+1 ;
        if (n==0)
                return A(m-1,1) ;
        return A(m-1, A(m, n-1)) ;
}

int main(){
        printf("A(2,1) = %d*******\n", A(2,1));
        printf("A(3,1) = %d*******\n", A(3,1));
        return (0);
}
```

L'appel A(2,1) nous donne : 5
L'appel A(3,1) nous donne : 13
La fonction d'Ackermann croît extrêmement rapidement : A(4,2) a déjà 19729 chiffres. Cette extrême croissance peut être exploitée pour montrer que la fonction f définie par $A(n,n)$ *croît plus rapidement que n'importe quelle fonction récursive primitive.*

Ackermann est cependant μ*-récursive. On le montre en utilisant une pile pour gérer la récursivité.*

3.5 Exercices

1. Nous souhaitons calculer dans la variable y, la valeur en x (réel) d'un polynôme de degré n. Les coeficients de ce polynôme sont des entiers mémorisés dans un tableau $intA[N+1]$. $y = a_0 + a_1x^1 + a_2x^2 \ldots . a_Nx^N$

   (a) Ecrire une fonction puiss qui calcule X^i

   (b) Ecrire la fonction récursive, poly, qui calcule le polynôme en utilisant la fonction puiss

   (c) Ecrire la fonctionn récursive polyAcc qui n'utilise pas , pour optimiser, la fonction puiss

   (d) Ecrire une version itérative

```
#include <stdio.h>
#define N 4
typedef int TAB[N+1];


float puiss(float X, int i)
{ if (i==0) return 1; else return X*puiss(X, i-1);}

/* fonction  recursive qui calcule un polynome de degre i*/
float poly (TAB A, float X, int i) {
        if ( i==0)
                return A[0];
        else
                return
                        (A[i]*puiss(X, i) + poly(A, X, i-1) );
}
/* fonction  recursive n'utilisant pas puiss pour faire le meme calcul*/
float PolyAcc(TAB A, float X, int i, float acc){
        if ( i==0)
                return A[0];
        else {
                acc = acc*X;
                return (  A[N-i+1]*acc +PolyAcc(A, X, i-1, acc) ) ;
        }
}
```

```
int main(){
TAB A = {1, 2, 5, 3, 2};
int i;
float X,y, p;
printf("taper une valeur entiere pour X=");
scanf ("%f", &X);
for (i=0; i<N+1; i++) printf("%d\n", A[i]);


printf("p=%f\n", poly(A,X,N));
printf("p=%f\n", PolyAcc(A,X,N, 1));

/* solution iterative n'utilisant pas puiss pour faire le meme calcul*/
i = 0;
p = A[0];
y = 1.0;

while (i < N)
{

        i = i + 1;
        y = y*X ;
        p = p + (A[i]*y);

}
printf("p=%f\n", p);

return 0;
}
```

2. Hyper

   (a) Donner les valeurs imprimées par le programme suivant et déduire de quoi s'agit-il.

   (b) Donner pour chaque printf..., le nombre d'appels de H.

   (c) Si la question b vous paraît difficile à traiter, modifier le programme précédent afin de compter le nombre d'appels récursifs effectués à chaque « printf ».

```
#include <stdio.h>
int H(int n, int a, int b) {
        if (n==0)
                return b+1;
        if (b==0)
                {
                if (n==1)
                        return a;
                if (n==2)
                        return 0;
                if (n>=3)
                        return 1;
        };
        return H(n-1, a, H(n, a, b-1));
}
int main(){
        printf("*****H=%d\n", H(1, 3, 3));
        printf("*****H=%d\n", H(2, 3, 3));
        printf("*****H=%d\n", H(3, 2, 3));
        printf("*****H=%d\n", H(4, 2, 3));
        return 0;
}
```

- *****H=6, 7 fois
- *****H=9 25 fois
- *****H=8 49 fois
- *****H=16 203 fois

la fonction int H(int n, int a, int b) calcule:

- a+b si n=1
- a*b si n=2
- a^b si n=3
- a à la puissance a b fois si n=4

```
#include <stdio.h>
int c;
int H(int n, int a, int b) {
    ++c;
    if (n==0)
        return b+1;
    if (b==0)
        {
        if (n==1)
            return a;
        if (n==2)
            return 0;
        if (n>=3)
            return 1;
    };
    return H(n-1, a, H(n, a, b-1));
}

int main(){
    c=0;
    printf("*****H=%d\n", H(1, 3, 3));
    printf("c=%d\n",c);
    c=0;
    printf("*****H=%d\n", H(2, 3, 3));
    printf("c=%d\n",c);
    c=0;
    printf("*****H=%d␣\n", H(3, 2, 3));
    printf("c=%d\n",c);
    c=0;
    printf("*****H=%d␣\n", H(4, 2, 3));
    printf("c=%d\n",c);
    return 0;
}
```

3. Ecrire une fonction récursive prenant en entrée un tableau de taille N. La fonction retourne le tableau inversé.

```
#include <stdio.h>
#include<stdlib.h>
typedef int * TAB;
void xch(int *a, int *b){
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
void inverser(TAB t, int i, int j){
    if (i<j){
        xch(&t[i], &t[j]);
        inverser(t, i+1, j-1);
    }
}
int main(){
    int N;
    TAB t;
    printf("N=");
    scanf("%d", &N);
    printf("\n");
    t=(int *) malloc(N*sizeof(int));
    for(int i=0; i<N; ++i){
        printf("t[%d]=  ",i);
        scanf("%d", &t[i]);

    }
    printf("\n");
    inverser(t,0,N-1);
    for(int i=0; i<N; ++i)
        printf("t[%d]=%d ",i, t[i]);
     printf("\n");
    return 0;
}
N=5
t[0]=1  t[1]=2  t[2]=3  t[3]=4  t[4]=5
t[0]=5  t[1]=4  t[2]=3  t[3]=2  t[4]=1
```

4. Ecrire une fonction récursive qui reçoit un entier naturel positif en entrée et retourne 1 si ce nombre est un palindrome et 0 sinon.

```
#include <stdio.h>
int palindrome(int i, int N, int acc){
    if (i==0){
        if(N!=acc)
            return 0;
        return 1;
    }
    return palindrome(i/10, N, 10*acc+ i%10);
}
int main(){
    int n;
    printf("n=");
    scanf("%d", &n);
    printf("palind=%d\n", palindrome(n,n,0));
    return 0;
}
```

```
n=123
palind=0
```

```
n=2552
palind=1
```

5. On souhaite calculer le nombre d'appels effectués lorsqu'on appele la fonction d'Ackerman. Modifier le programme ci-dessus afin de pouvoir calculer et retourner en plus de sa valeur, son nombre d'appel.

```
#include <stdio.h>
int A(int m,int n){
        if (m==0)
                return n+1 ;
        if (n==0)
                return A(m-1,1) ;
        return A(m-1, A(m, n-1)) ;
}
```

Solution 1

```
#include <stdio.h>
int A(int m,int n, int * c){
    ++ (*c);
    if (m==0)
        return n+1 ;
    if (n==0)
        return A(m-1,1,c) ;
    return A(m-1, A(m, n-1,c),c) ;
    }
int main(){
    int c;
    c=0;
    printf("A(2,1) = %d ", A(2,1,&c));
    printf("nombre d'appels=%d\n", c);
    c=0;
    printf("A(3,1) = %d \n", A(3,1, &c));
    printf("nombre d'appels=%d\n", c);
    c=0;
    printf("A(3,1) = %d \n", A(3,2, &c));
    printf("nombre d'appels=%d\n", c);
    return (0);
}
A(2,1) = 5 nombre d'appels=14
A(3,1) = 13 nombre d'appels=106
A(3,1) = 29 nombre d'appels=541
```

Solution 2

```
#include <stdio.h>
int c;
int A(int m,int n){
    ++c;
    if (m==0)
        return n+1 ;
    if (n==0)
        return A(m-1,1) ;
    return A(m-1, A(m, n-1)) ;
    }
int main(){
    c=0;
    printf("A(2,1) = %d\n", A(2,1));
    printf("nombre d'appels=%d\n", c);
    c=0;
    printf("A(3,1) = %d\n", A(3,1));
    printf("nombre d'appels=%d\n", c);
    c=0;
    printf("A(3,2) = %d\n", A(3,2));
    printf("nombre d'appels=%d\n", c);

    return (0);
}
```

6. Ecrire une fonction récursive paramétrée par n permettant d'afficher, pour $n = 5$ le schéma suivant

```
*****
****
***
**
*
**
***
****
*****
```

```
void BHK(int n, int N){
        if (n>0){
                for (int i=0; i<n; ++i)
                        printf("*");
                printf("\n");
                BHK(n-1, N);
                if (n<N){
                        for (int i=0; i<=n; ++i)
                                printf("*");
                        printf("\n");
                }
        }
}
```

7. Ecrire une fonction récursive qui calcule le nombre de valeurs d'un tableau de taille $N \geq 0$ qui sont supérieures à la moyenne entière des valeurs de ce tableau.

```
#include <stdio.h>
#include <stdlib.h>
typedef int * TAB;
typedef struct  {
  int mc;
  int nbsup;
} couple;
couple NbSupMoy(TAB L, int d,int s, int n){
  couple p;
  if (d==n) {
    if (n==0){
      p.mc = 0;
      p.nbsup = 0;
    }
    else {
      p.mc = s/n;
      p.nbsup = n;
    }
    return p;
  }
    else {
      p = NbSupMoy(L, d+1, s+L[d],n);
      if (L[d]<= p.mc){
        --p.nbsup;
        return p;
      }
    }
  return p;
}
int main(){
  int n;   couple p;
  TAB T=(int *)malloc(n*sizeof(int));
  printf("n=");
  scanf("%d", &n);
  for(int i=0; i<n; ++i)
    scanf("%d", &T[i]);
  p = NbSupMoy(T, 0 , 0, n);
  printf("mc=%d, nbsup=%d", p.mc, p.nbsup);
  return 0;
}
```