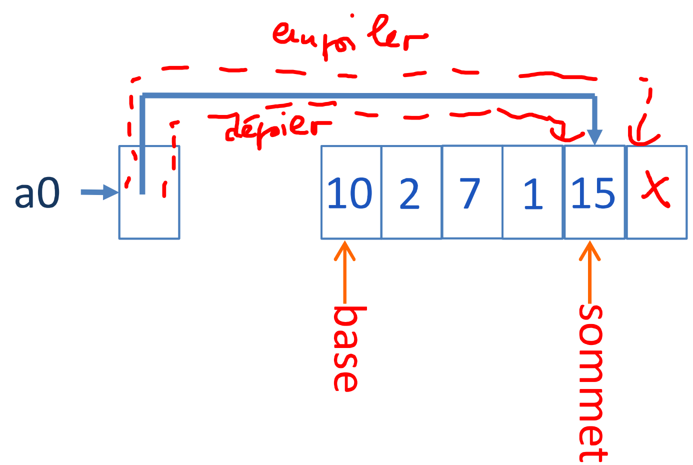
II.5.4 Les piles (P. 13)

- Définition

Une pile est une liste dans laquelle les insertions et les suppressions se font uniquement en tête (au sommet). LIFO ou DAPS

- Représentation contigüe
- Opérations de base
  - empiler (push) / dépiler (pop)



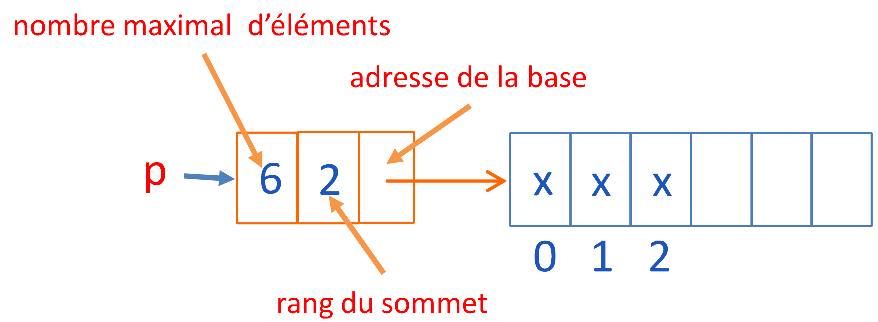
Gestion du pointeur de tête:

Empiler: m(a0):=cm(a0)+1 ou (+k)
mettre en place la valeur

sauver la valeur qui est sommet
Dépiler: m(a0):=cm(a0)-1 ou (~~+~~k)

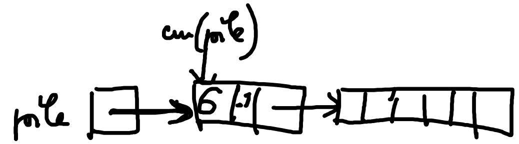
- Applications
  - Compilateurs/interpréteurs (exemple: évaluation d’expressions arithmétiques)
  - Parcours d’arbres
  - Suppression de la récursivité des appels non terminaux des sous-programmes récursifs

- Structure d’une pile
  - la base (l’adresse du début de la liste)
  - le sommet (le rang de l’élément au sommet)
  - la taille(le nombre maximal d’éléments)
- Gestion d’une pile
  - initialiser / libérer la pile
  - tests pile vide / pile pleine
  - Empiler, dépiler et sommet

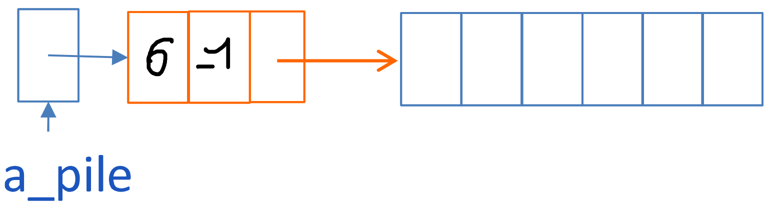


Opération élémentaires d’un gestionnaire de pile

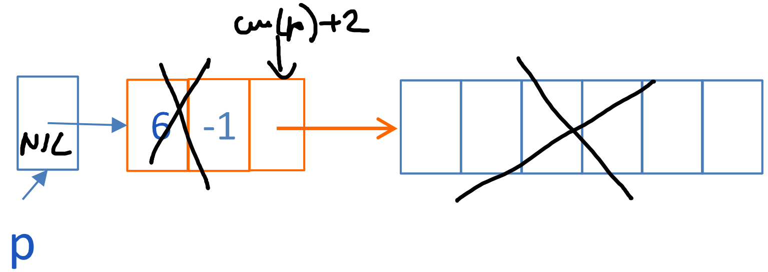
Fonction initPile(taille):
    m(pile) := alloc(3); [allocation du bloc de tête]
    Si cm(pile)≠NIL **alors [allocation réussie]**
        m(cm(pile)) := taille; [taille max]
        m(cm(pile) + 1) := -1; [la pile est vide]
        m(cm(pile) + 2) := alloc(taille); [allocation de la pile]
        Si (cm(cm(pile)+2)=NIL) **alors [échec allocation]**
            libérer(cm(pile));
            m(pile) := NIL;
        fsi;
    fsi;
    retourner (cm(pile));
Fin;



Un appel: m(ma_pile):= initPile(6);

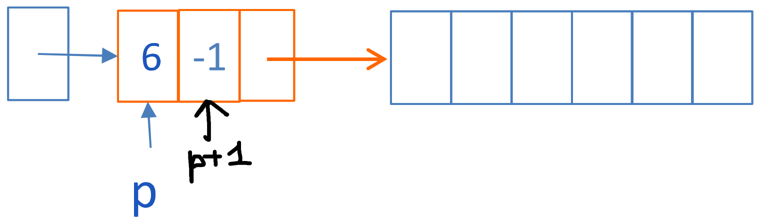


```
Procédure libérerPile(ES: p ):
    libérer(cm(cm(p)+2));
    libérer(cm(p));
    m(p) := NIL;
Fin;
```



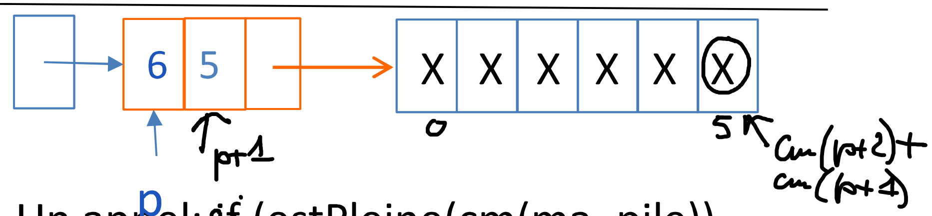
Un appel: libérerPile(ma_pile);

```
Fonction estVide(p):
    retourner (cm(p+1)) = -1);
Fin;
```



Un appel: if(estVide(cm(ma_pile))….

```
Fonction estPleine(p):
    retourner (cm(p+1) = cm(p)-1);
Fin;
```



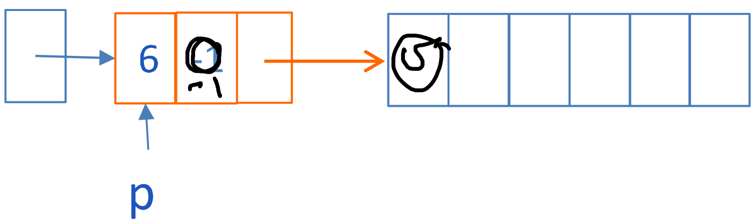
Un appel: ~~si~~ if(estPleine(cm(ma_pile))….

```
Procédure sommet (E:p; s:adrV, état):
    m(état) := 1;
    Si NON estVide(p) alors
        m(adrV) := cm(cm(p+2) + cm(p+1));
        m(état):=0;
    fsi;
Fin;
```

un appel:
sommet(cm(ma_pile), res,retour);
If (retour)…..

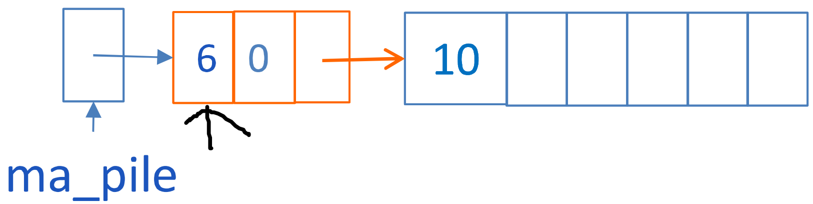
```
fonction empiler(E:p,v):
    m(état) := 1;
    Si NON estPleine(p) alors
        m(p+1) := cm(p+1) + 1;
        m(cm(p+2) + cm(p+1)) := v;
        m(état) := 0;
    fsi;
retourner cm(état);
Fin;
```

m(état)=0 : réussi
m(état)=1 : échec

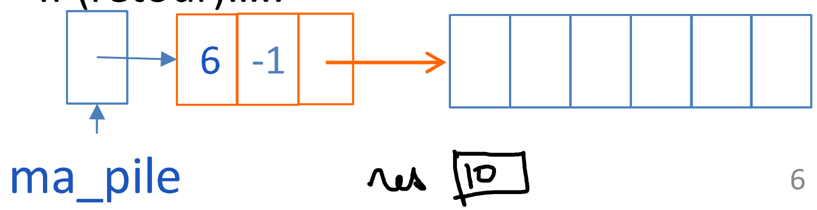



```
Procédure dépiler(E:p; s:adrV, état):
    m(état) := 1;
    Si NON estVide(p) alors
        m(adrV) := cm(cm(p+2) + cm(p+1));
        m(p+1) := cm(p+1) - 1;
        m(état) := 0;
    fsi;
Fin;
```

un appel:
m(retour):=empiler(cm(ma_pile), 10);



un appel:
dépiler(cm(ma_pile), res,retour);
If (retour).....



II.5.4 Les piles – Utilisation

- Application aux procédures récursives
  - Appels terminaux

```
Fonction dichoRecur(a, b , u):
    Si a=b alors  retourner a;
    Sinon
        m(mil) := (a+b) div 2;
        Si v <= cm(cm(mil)) alors
            retourner dichoRecur(a, cm(mil), u);
        Sinon
            retourner dichoRecur(cm(mil)+1, b, u);
        fsi;
    fsi;
Fin;
```

Pas de traitement après l'appel

  - Appels non terminaux

```
Procédure rec(x, y)
    …
    rec(x1, y1);
    …
Fin
```

appel non terminal

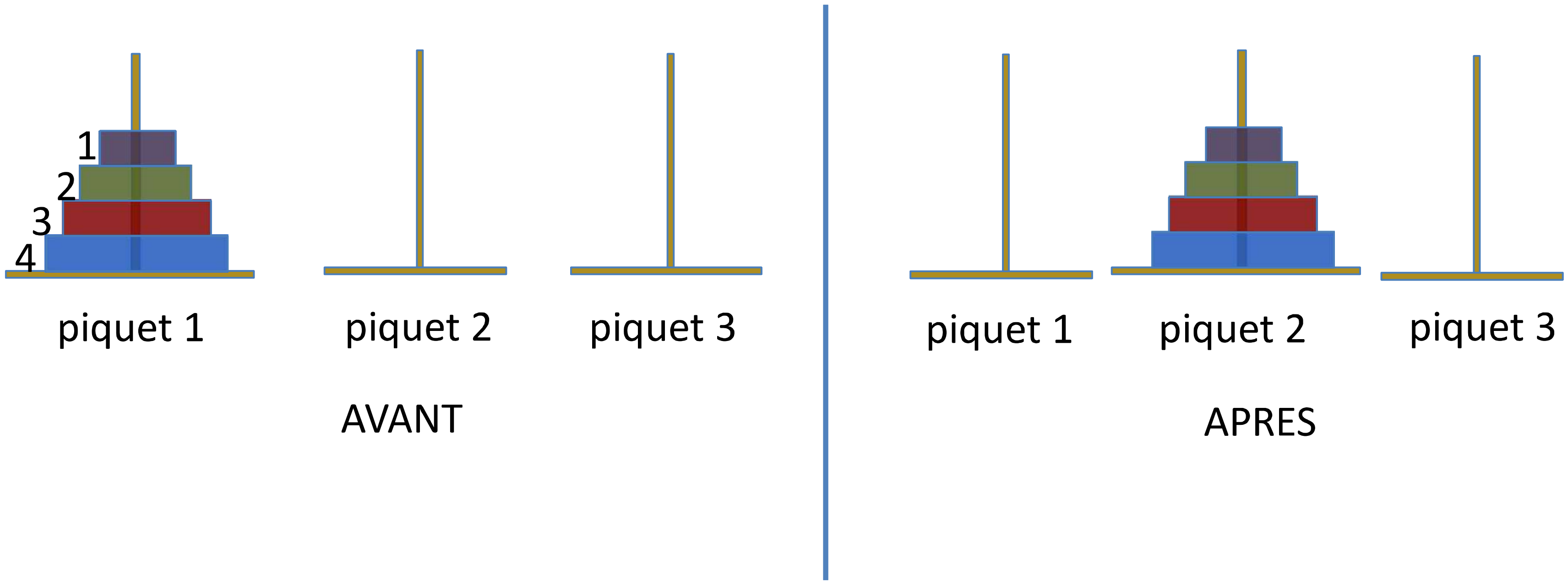
traitements après l'appel récursif

- Suppression de la récursivité: principe
    1. Faire une trace de l’exécution sur un exemple; (1)
    2. Faire l’organigramme de la procédure récursive (2)

    Analyser la trace et l’organigramme pour donner les caractéristiques des appels (appel terminal/non terminal, les éléments à empiler, évolution des résultats)

    3. Suppression des appels terminaux (3)
    4. Suppression des appels non terminaux (4)
    5. Ecriture de la procédure itérative (5)

- Application au jeu des tours de Hanoï
  - Transfert des pièces du piquet 1 vers le piquet 2, le piquet 3 servant d’intermédiaire
  - Règles de déplacement : déplacer
    - ✧ une pièce à la fois
    - ✧ une pièce sur une pièce plus grande ou sur une place vide



- Tours de Hanoï – solution récursive

```
Procédure Hanoï(E: n, d, a)
    Si n≠0 alors
        Hanoï(n-1, d, 6-d-a)
        écrire(n, d, a);            [Déplacement de la pièce n de d à a]
        Hanoï(n-1, 6-d-a, a);
    fait;
Fin
```

Lexique :
- n : nombre de pièces à déplacer (en fait n° de la pièce à déplacer)
- d : n° du piquet de départ
- a : n° du piquet d'arrivée

Trace pour Hanoï(3,1,2)

Procédure Hanoï(E: n, d, a)

Si n≠0 **alors**

→ Hanoï(n-1, d, 6-d-a)

écrire(n, d, a);

Hanoï(n-1, 6-d-a, a);

fait;

Fin

H(3,1,2) =

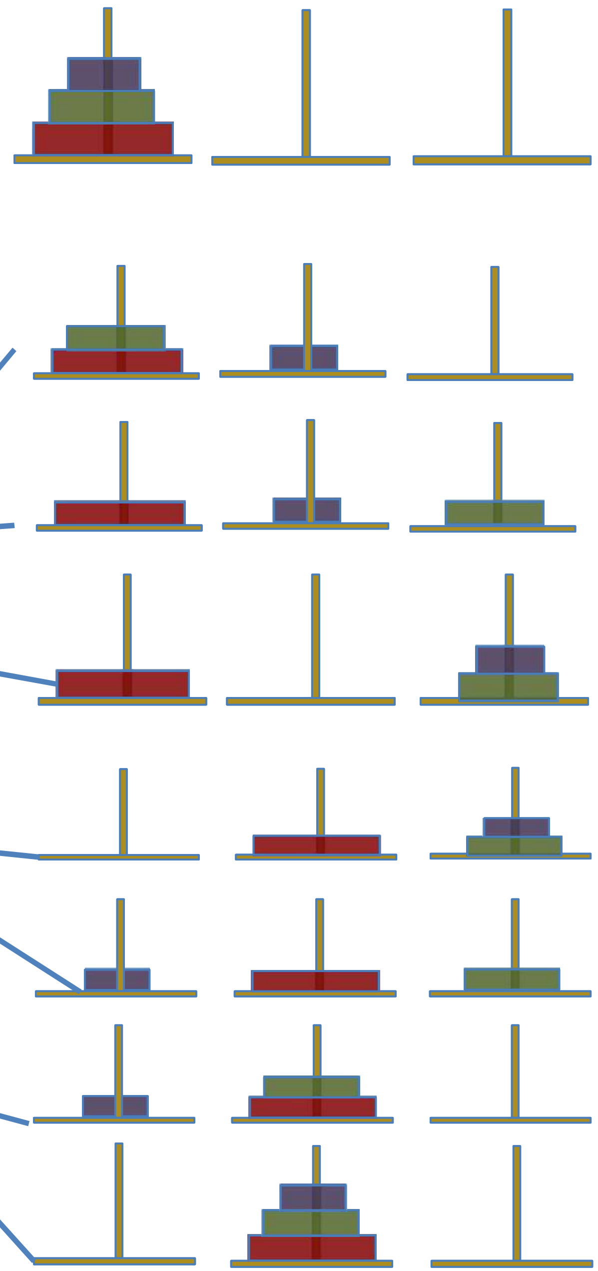
- H(2,1,3)
  - H(1,1,2)
    - H(0,1,3) n=0
    - écriture(1,1,2)
    - H(0,3,2) n=0
  - écriture(2,1,3)
  - H(1,2,3) etc ……

Etape 1 :TRACE avec n=3

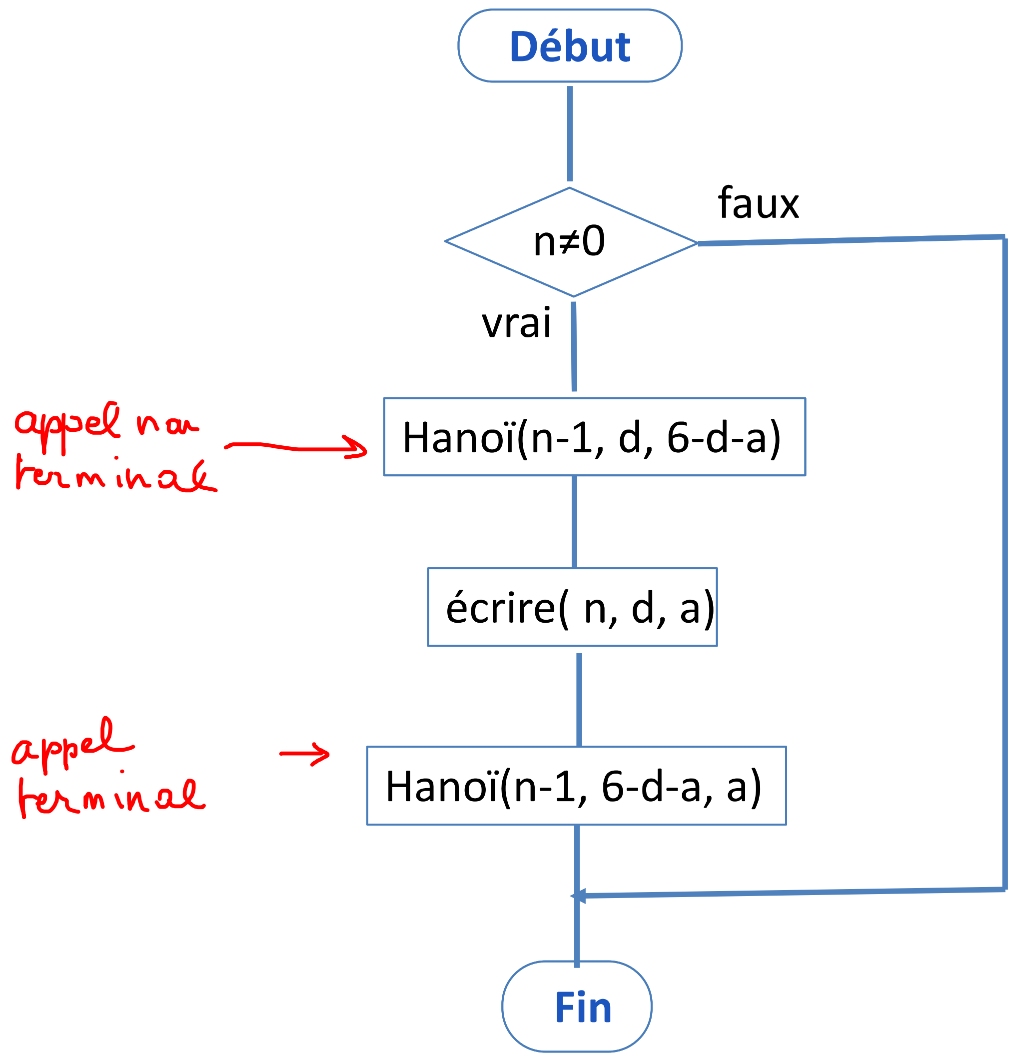
```
Procédure Hanoï(E: n, d, a)
    Si n≠0 alors
        Hanoï(n-1, d, 6-d-a)
        écrire(n, d, a);
        Hanoï(n-1, 6-d-a, a);
    fait;
Fin
```

Hanoï(3, 1, 2)

- Hanoï(2, 1, 3)
  - Hanoï(1, 1, 2)
    - Hanoï(0, 1, 3)
    - écrire(1, 1, 2)
    - Hanoï(0, 3, 2)
  - écrire(2, 1, 3)
  - Hanoï(1, 2, 3)
    - Hanoï(0, 2, 1)
    - écrire(1, 2, 3)
    - Hanoï(0, 1, 3)
- écrire(3, 1, 2)
- Hanoï(2, 3, 2)
  - Hanoï(1, 3, 1)
    - Hanoï(0, 3, 2)
    - écrire(1, 3, 1)
    - Hanoï(0, 2, 1)
  - écrire(2, 3, 2)
  - Hanoï(1, 1, 2)
    - Hanoï(0, 1, 3)
    - écrire(1, 1, 2)
    - Hanoï(0, 3, 1)



Etape 2 : Organigramme - récursif

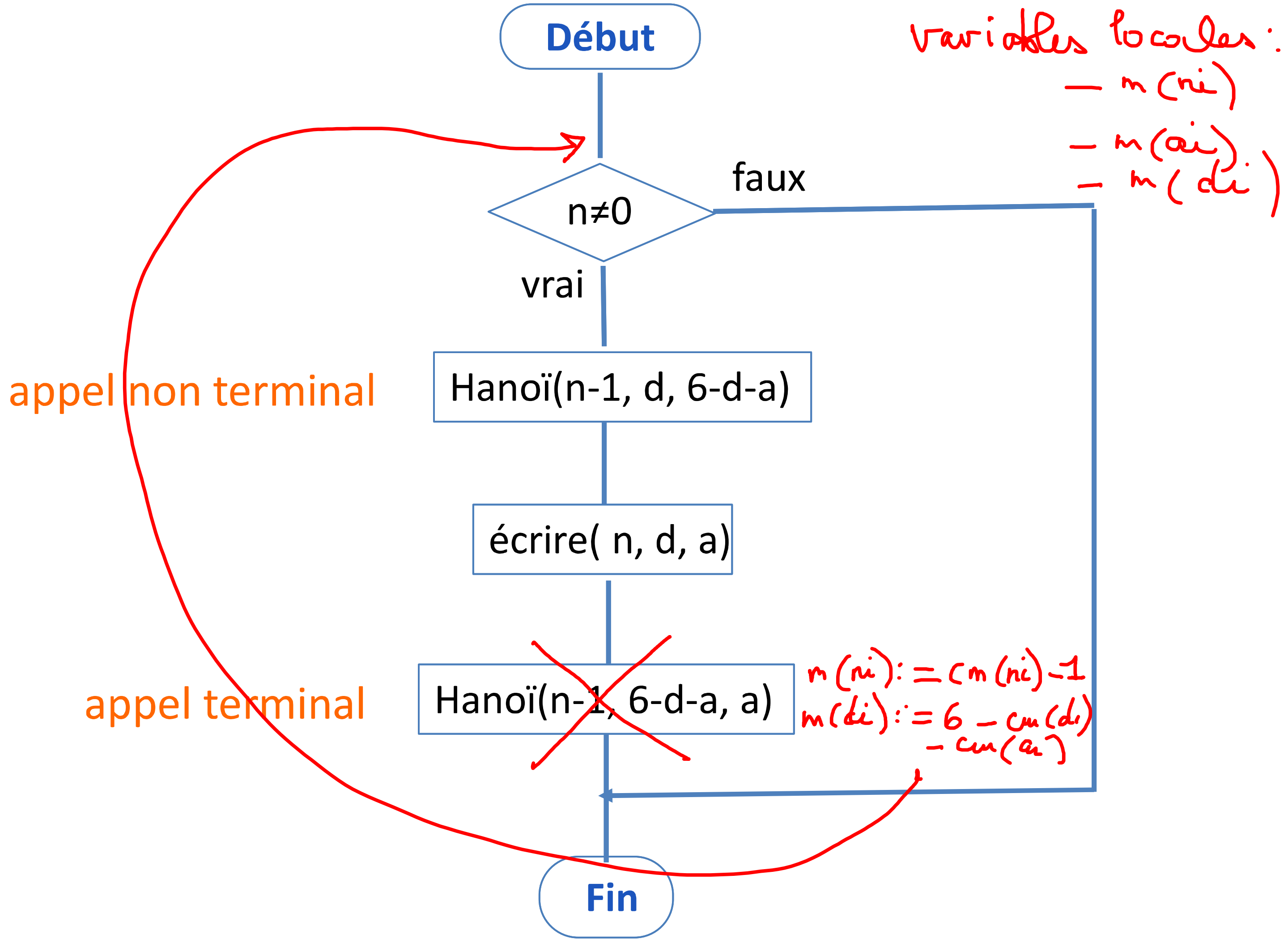


Etape 3 : Suppression des appels terminaux

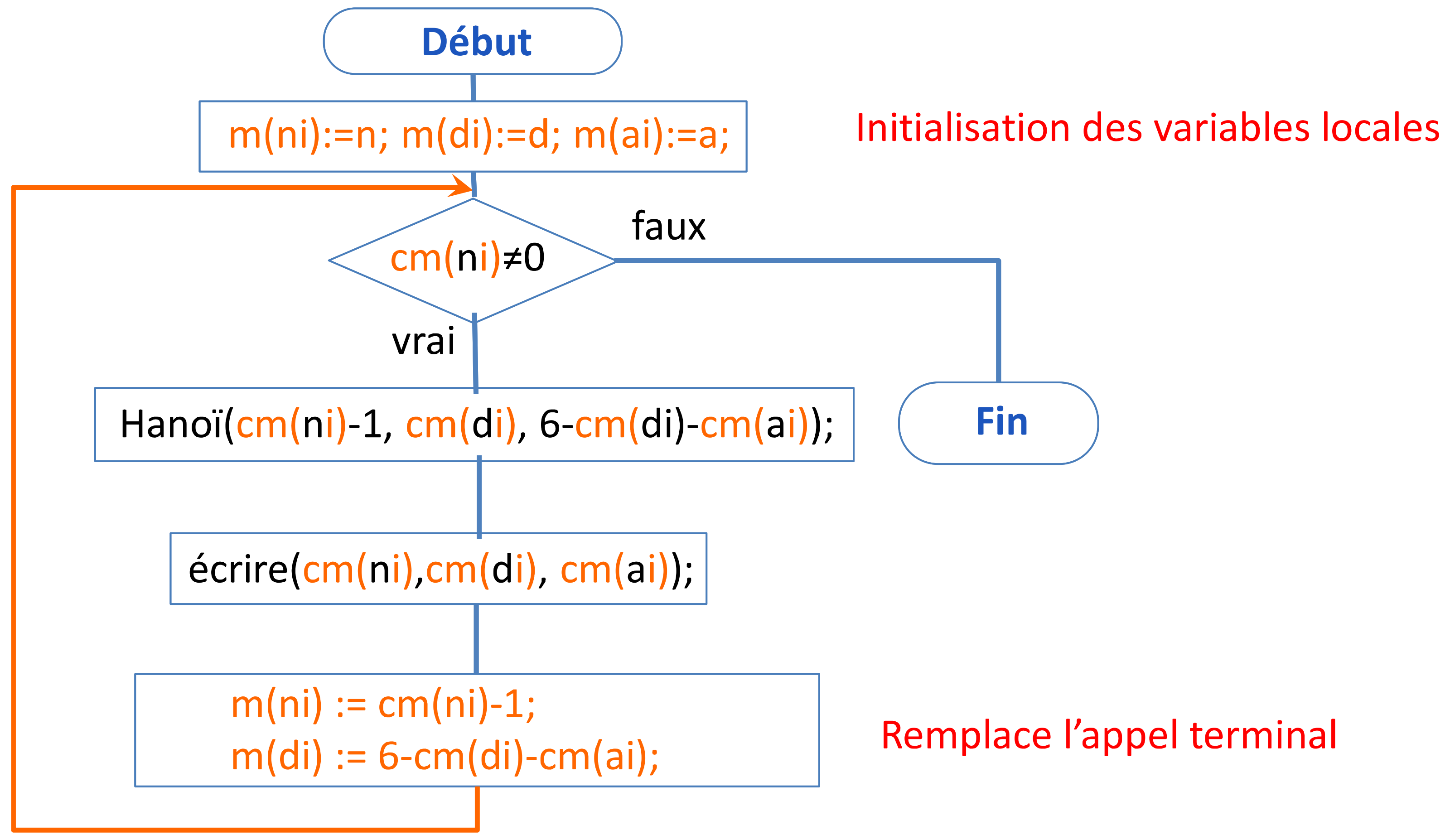
Application des principes vus pour la dichotomie:

- créer une variable locale par paramètre modifié;
- Remplacer l’appel par une modification des variables locales et un retour au début

Etape 3 : Suppression des appels terminaux

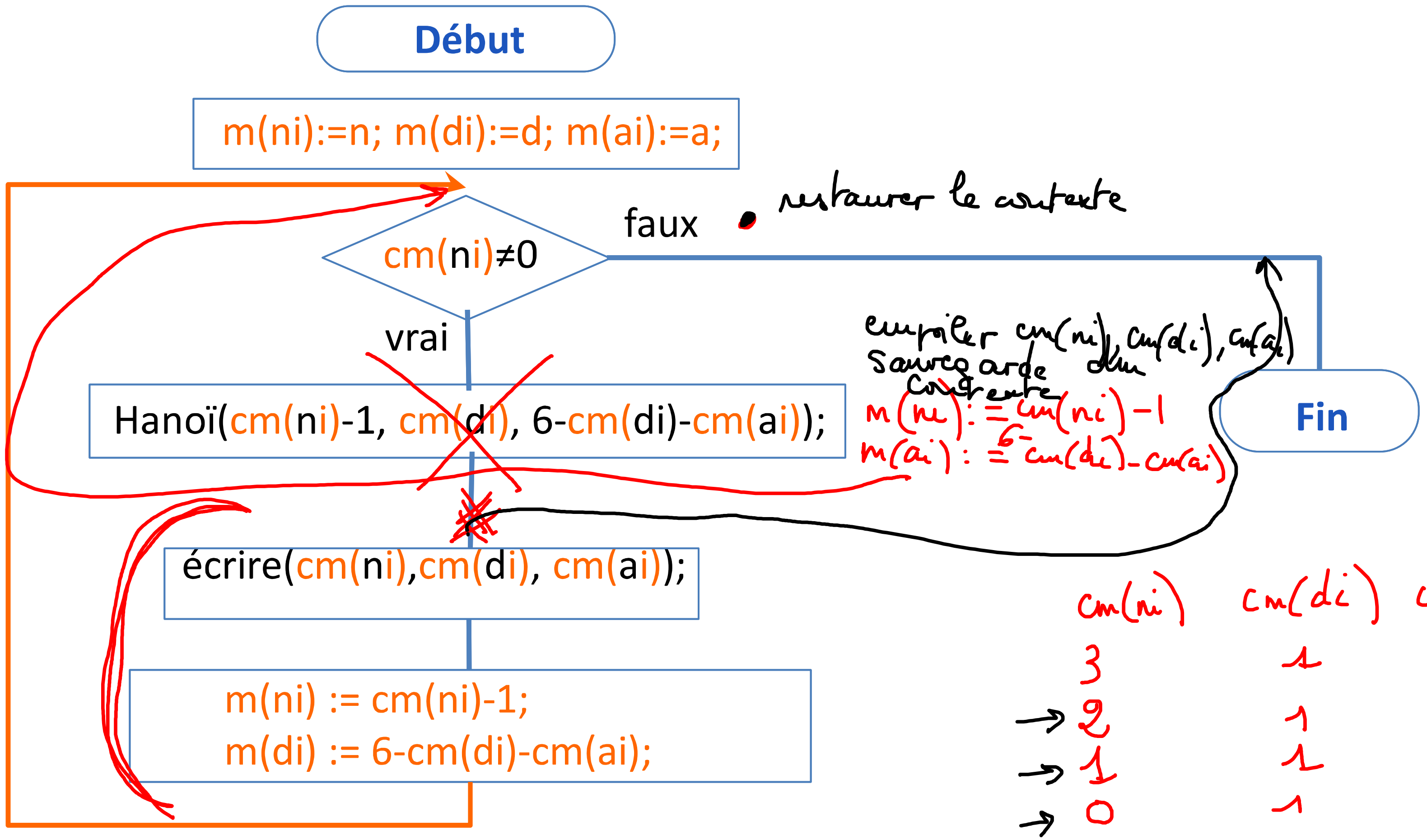


Etape 3 : Suppression des appels terminaux

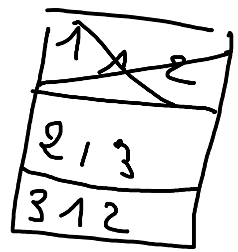


Le retour au test permet d'exécuter à nouveau l'algorithme (comme le faisait l'appel terminal)

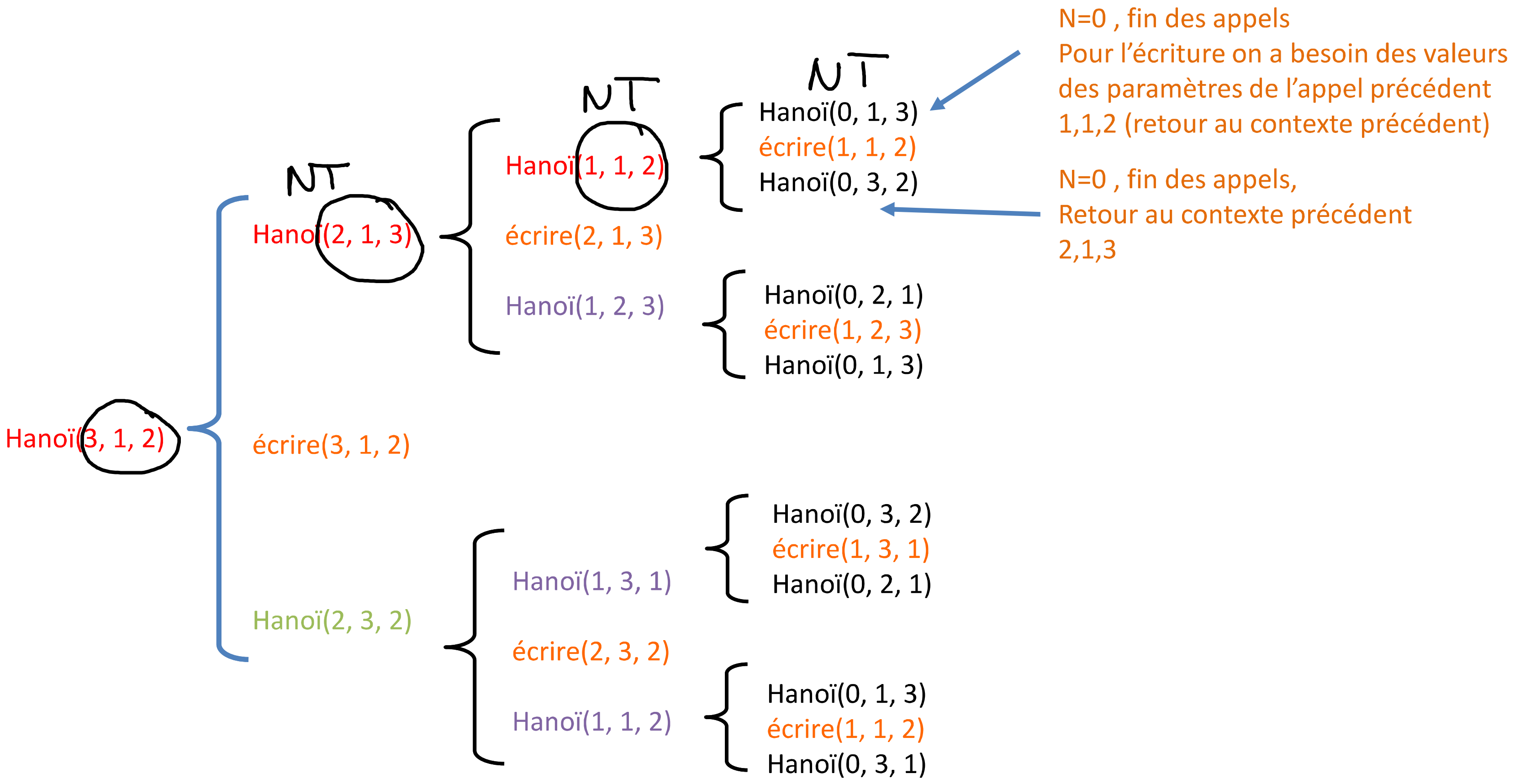
Etape 4 : Suppression des appels non terminaux



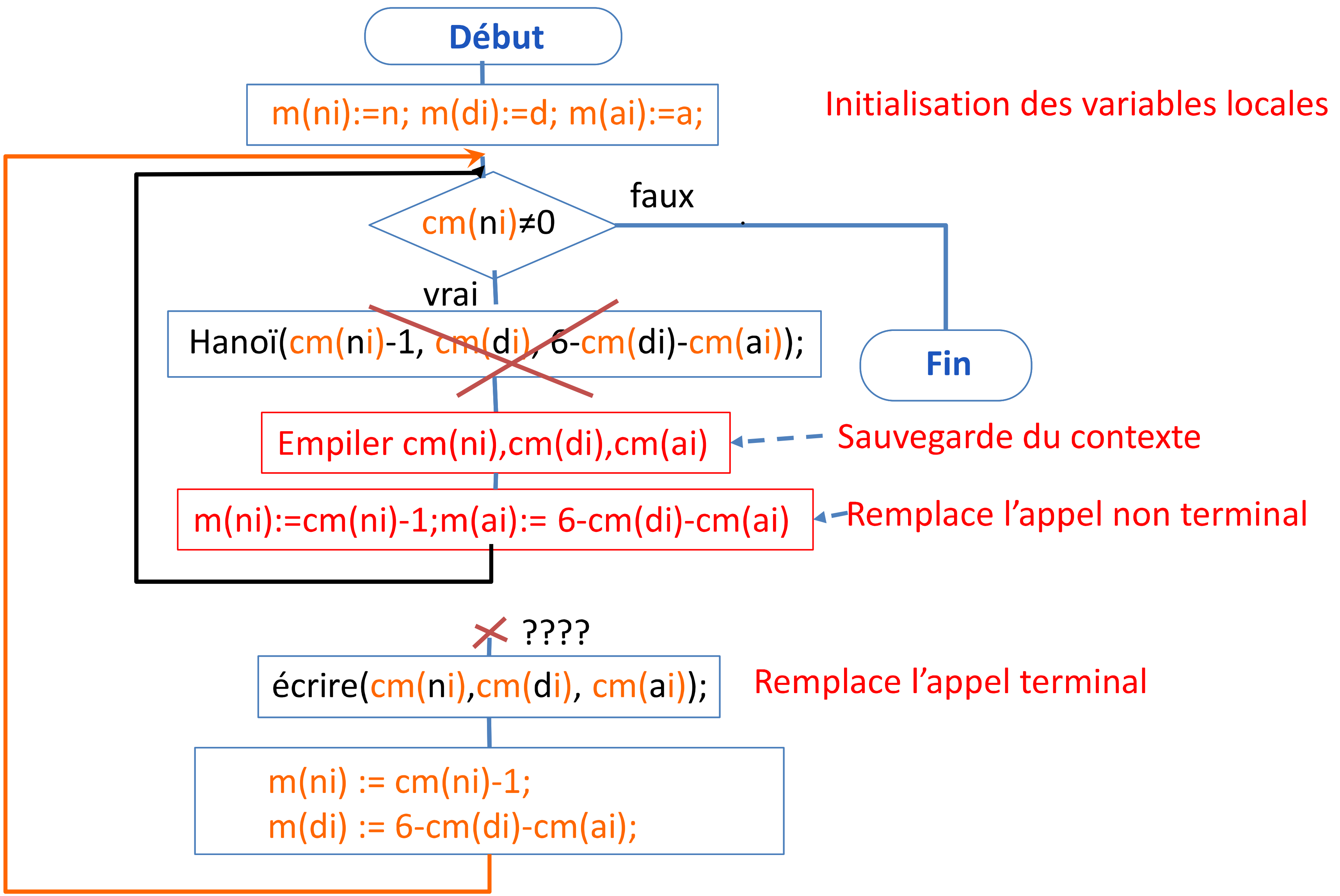
cm(ni)	cm(di)	cm(ai)
3	1	2
→ 2	1	3
→ 1	1	2
→ 0	1	3



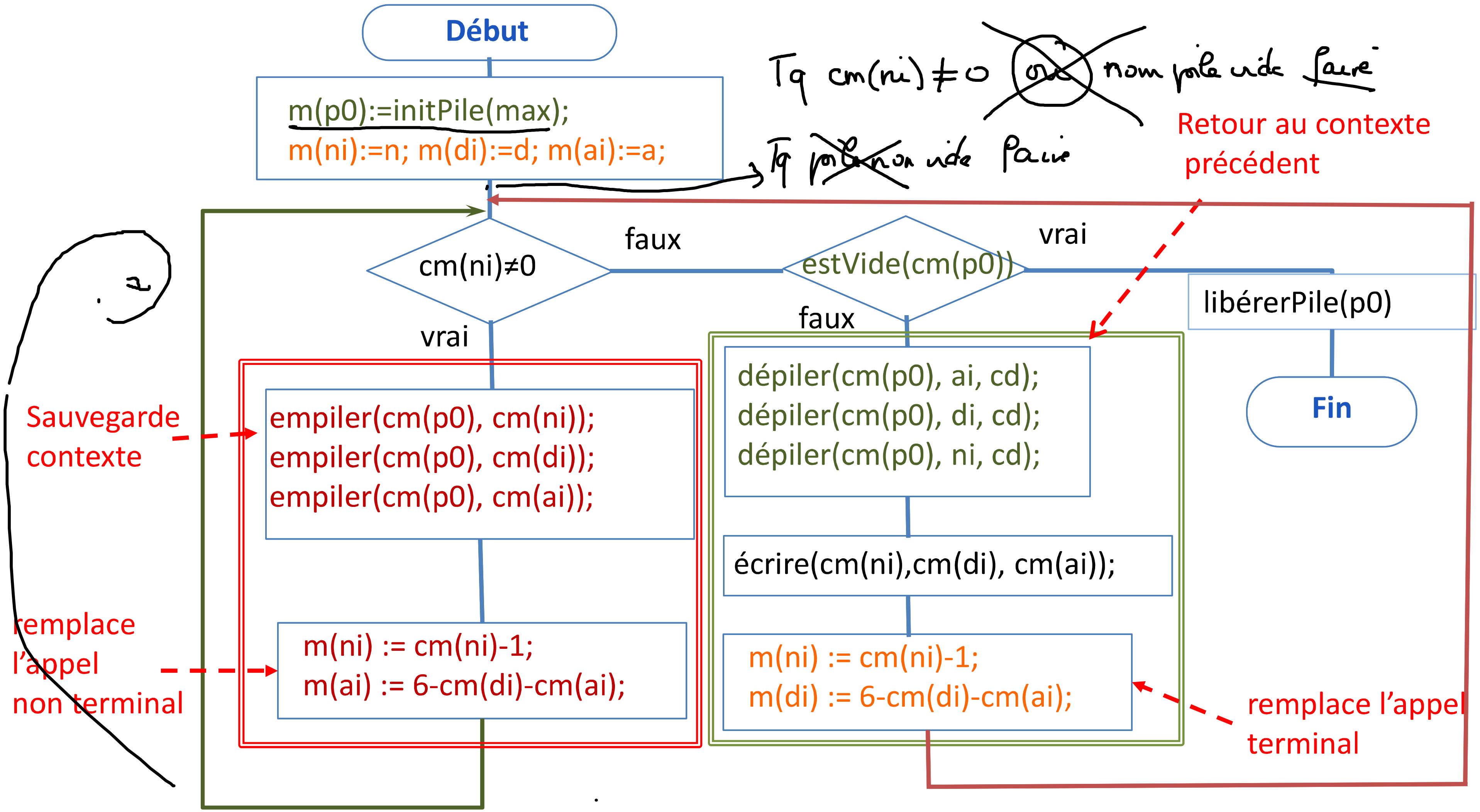
Etape 3 : Suppression des appels non terminaux



Etape 4 : Suppression des appels non terminaux



Etape 4 : Suppression des appels non terminaux



cd : adresse du code de retour

Etape 5 : Ecriture de la procédure itérative

```
Procédure HanoïIter(E : n, d, a):
    m(p0) := initPile(MAX);
    m(ni) := n; m(di) := d; m(ai) := a;
    m(fin) := Faux;
    Tant que NON cm(fin) faire
        Tant que cm(ni)≠0 faire
            empiler(cm(p0), cm(ni));     } Sauvegarde du contexte avant l'appel non terminal
            empiler(cm(p0), cm(di));
            empiler(cm(p0), cm(ai));
            m(ni) := cm(ni)-1;           } Remplace l'appel non terminal
            m(ai) := 6-cm(di)-cm(ai);
        fait;
     Si NON estVide(cm(p0)) alors
          dépiler(cm(p0), ai, cd);       } Restauration du contexte
          dépiler(cm(p0), di, cd);
          dépiler(cm(p0), ni, cd);
          écrire(cm(ni), cm(di), cm(ai));
          m(ni) := cm(ni)-1;             } Remplace l'appel terminal
          m(di) := 6-cm(di)-cm(ai);
     Sinon
          m(fin) := Vrai;
     fsi;
     fait;
     libérerPile(p0);
Fin;
```

Autre exemple: Fonction d'Ackermann

Hypothèses : m, n entiers ≥ 0

$$Ack(m,n) = \begin{cases} n+1, & \text{si } m=0 \\ Ack(m-1,1), & \text{si } m>0 \text{ et } n=0 \\ Ack(m-1, Ack(m,n-1)), & \text{si } m>0 \text{ et } n>0 \end{cases}$$

```
Fonction Ack(m,n):
   Si m=0 alors retourner n+1;
   Sinon
      Si n=0 alors retourner Ack(m-1,1);
      Sinon retourner Ack(m-1, Ack(m,n-1));
      fsi;
   fsi;
Fin;
```

- Fonction d’Ackermann

Values of $A(m, n)$

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$

Faire le début de la trace à la main

Trace pour : $m=2$, $n=1$

$$Ack(m,n)=\begin{cases} n+1, & \text{si } m=0 \\ Ack(m-1,1), & \text{si } m>0 \text{ et } n=0 \\ Ack(m-1, Ack(m,n-1)), & \text{sinon} \end{cases}$$

$Ack(2,1) = Ack(1, \underbrace{Ack(2,0)}_{=3})$

$\hookrightarrow Ack(1,1) \rightarrow \underbrace{Ack(0, \underbrace{Ack(1,0)}_{=2})}_{=3}$

$\hookrightarrow Ack(0,1) = 2$

$Ack(0,2) = 3$

$Ack(1,1) = 3$

$Ack(1,3) = Ack(0, \underbrace{Ack(1,2)}_{=4})$

$\hookrightarrow Ack(0, \underbrace{Ack(1,1)}_{=3})$

$\hookrightarrow Ack(0, \underbrace{Ack(1,0)}_{=2})$

$\hookrightarrow Ack(0,1) = 2$

$Ack(0,2) = 3$

$Ack(0,3) = 4$

$Ack(0,4) = 5$

$Ack(1,3) = 5$

$Ack(2,1) = 5$

- Fonction d'Ackermann - Trace de Ack(2, 1)

$$Ack(m,n)=\begin{cases} n+1, & \text{si } m=0 \\ Ack(m-1,1), & \text{si } m>0 \text{ et } n=0 \\ Ack(m-1, Ack(m,n-1)), & \text{sinon} \end{cases}$$

Ack(2,1) = Ack(1,Ack(2,0))

Ack(1,1)

Ack(0,Ack(1,0))

Ack(0,1)=2

Ack(0,2) = 3

= 5

Ack(1,3) = Ack(0, Ack(1,2))

Ack(0,Ack(1,1))

Ack(0,Ack(1,0))

Ack(0,1) = 2

Ack(0,2)=3

Ack(0,3)=4

Ack(0,4) = 5