Les fonctions [1]

Salem BENFERHAT

Centre de Recherche en Informatique de Lens (CRIL-CNRS)
email : benferhat@cril.fr

[1] Version préliminaire du cours. Tout retour sur la forme comme sur le fond est le bienvenu.

Premier contrôle continu

La semaine du 13 octobre

ou

La semaine du 20 octobre

Points bonus

pendant

les TPs

Comprendre l'IA sans formules mathématiques

- Mercredi 8 octobre - 17h30 > 19h
- Atrium du Louvre-Lens Vallée - 84 rue Paul Bert - 62300 LENS
- Les réseaux de neurones à travers une histoire ...

La programmation modulaire

- En C, les sous-programmes sont appelées des fonctions.
- La décomposition d'un programme en fonctions offre plusieurs avantages :
  - Améliorer la lisibilité des programmes.
  - Faciliter la détection d'erreurs. Chaque fonction peut-être vérifiée et testée séparément.
  - Eviter la duplication des codes.

La programmation modulaire

- Un autre avantage est la possibilité de ré-utiliser des codes pour d'autres programmes.
  - Par exemple, vous pouvez écrie une fonction qui vérifie si un nombre est premier ou non.
  - Puis utiliser cette fonction dans différents programmes.
  - D'ailleurs c'est le principe des bibliothèques C.
  - Par exemple, les fonctions :
    - printf,
    - scanf,
    - etc.

    sont utilisés dans différents programmes.

Les fonctions

Une fonction dispose :

- d'un identifiant.
- Des déclarations de paramètres ou d'argments.
- Le type de la valeur retournée.
- Le corps de la fonction.

Les fonctions : plus précisément

La syntaxe d'une fonction est :

Type_de_la_fonction identifiant (arg$_1$, arg$_2$, ...)
{
    Corps de la fonction [avec un return]
}

- La convention pour la définition des noms de fonctions est la même que celle utilisée pour les variables.
- La valeur retournée peut-être :
  - de type quelconque, y compris :
    - les pointeurs, et
    - les types complexes que nous verrons plus tard.
- Lorsque la fonction ne retourne aucune valeur, on utilisera le mot clef "void".

Les fonctions : plus précisément

La syntaxe d'une fonction est :

Type_de_la_fonction identifiant (arg$_1$, arg$_2$, ...)
{
    Corps de la fonction [avec un return]
}

- Chaque argument (appelé aussi paramètre d'entrée) est simplement déclaré sous la forme :

    type_argument *identifiant_argument*

- Lorsqu'aucun argument n'est utilisé, le mot clef "void" est utilisé.
- Ce mot clef "void" précise, explicitement, que la fonction ne contient pas d'arguments

Les fonctions : plus précisément

La syntaxe d'une fonction est :

Type_de_la_fonction identifiant (arg$_1$, arg$_2$, ...)
{
    Corps de la fonction [avec un return]
}

Le corps de la fonction contient :

- des déclarations de variables, dites locales.
- Une suite d'instructions qui contient, au moins une fois, l'expression :

return valeur (*ou expression*);

- La valeur retournée est de même type que celui de la fonction.
- Bien sûr, si le type retournée est "void", alors l'instruction "return" ne doit pas être utilisée.

Prenons
des exemples

Un premier exemple

Exemple de fonctions avec un seul argument

- Ecrire une fonction C qui :
  - qui prend en entrée un entier relatif a (positif ou négatif) et
  - retourne sa valeur absolue.

Un premier exemple

Commençons par l'entête

- Il nous faut un nom pour identifier la fonction.
  - Soyons originaux, appelons cette fonction : absolu.
- Ensuite, il faut déclarer les paramètres ou les arguments.
  - Dans notre exemple, nous avons un seul paramètre à déclarer :

$$int \quad a$$

- Finalement, il faut préciser le type de retour. Ca sera :
  - type "int"
  - on peut aussi utiliser "unsigned int".

Un premier exemple

Commençons par l'entête

Pour résumer, l'entête de la fonction est :

```
2 int absolu (int a)
```

Un premier exemple

Corps de la fonction

- Il nous reste maintenant à spécifier le corps de la fonction.
- Nous n'avons besoin d'aucune variable locale pour réaliser l'objectif de la fonction.
- Comme notre fonction est de type int, il faudra alors qu'elle contienne au moins une instruction qui retournerait une valeur (ou une expression) de type int.

Un premier exemple

Voici donc un exemple d'écriture de notre fonction "absolu" :

```
1 int absolu (int a)
2 {
3  if (a<0)
4    return -a;
5    else return a;
6 }
```

Un premier exemple

Donner une écriture de notre fonction ”absolu” sans utiliser le ”else”.

Un premier exemple

Voici donc un autre exemple d'écriture de notre fonction "absolu" (sans utiliser le "else"):

```
int absolu (int a)
{
 if (a<0)
  return -a;
 return a;
}
```

Un premier exemple

Voici donc un autre exemple d'écriture de notre fonction "absolu" (sans utiliser le "else"):

```
1 int absolu (int a)
2 {
3  if (a<0)
4   return -a;
5  return a;
6 }
```

- En effet, le else n'est pas obligatoire.

Un premier exemple

Voici donc un autre exemple d'écriture de notre fonction "absolu" (sans utiliser le "else"):

```
1 int absolu (int a)
2 {
3  if (a<0)
4   return -a;
5  return a;
6 }
```

- En effet, le else n'est pas obligatoire.
- Car, en langage C, le "return" provoque la fin de l'exécution de la fonction.

Un premier exemple

Donner une écriture de notre fonction ”absolu” sans utiliser le ”if”.

Un premier exemple

Voici encore une autre écriture de notre fonction ”absolu” :

```
1 int absolu (int a)
2 {
3  return (a<0) ? -a: a;
4 }
```

Un premier exemple

Voici encore une autre écriture de notre fonction ”absolu” :

```
1 int absolu (int a)
2 {
3  return (a<0) ? -a: a;
4 }
```

- Ici on utilise le conditionnement ternaire.

Un autre exemple

Exemple de fonctions avec plusieurs arguments

- Ecrire une fonction C qui :
  - qui prend en entrée deux entiers a et b, et
  - retourne la valeur maximale de a et b.

Un autre exemple

Solution

```
int max (int a, int b)
{
 if (a>=b)
   return a;
   else return b;
}
```

Un autre exemple

Solution

```
int max (int a, int b)
{
 if (a>=b)
   return a;
   else return b;
}
```

Remarques

- Le ”else” n’est pas obligatoire et

Un autre exemple

Solution

```
int max (int a, int b)
{
 if (a>=b)
   return a;
   else return b;
}
```

Remarques

- Le ”else” n’est pas obligatoire et
- on peut aussi utiliser le conditionnement ternaire.

Utilisation des fonctions

- Une fois une fonction est définie (écrite), elle peut-être utilisée aussi bien dans le programme principale que dans le corps d'une autre fonction.
- On parle alors d'appels de fonctions.
- Dans un premier temps, on se contentera de recopier son entête dans les fonctions (comme le main) où elle sera utilisée.

Utilisation des fonctions

- La valeur retournée peut-être
  - affichée (imprimée),
  - affectée à une variable ou
  - utilisée dans une expression.
- Bien sûr, chaque argument doit-être au préalable renseigné (par une expression ou une valeur).

Exemples d'utilisation des fonctions

La fonction "absolu"

L'exemple suivant montre l'appel à la fonction "absolu" par l'instruction "printf" au niveau du programme principal.

```
int main (void)
{
    int absolu (int a);
    int i=-20;
    printf ("La valeur absolue de i est : %d.\n ", absolu(i));
  return(0);
}
```

Exemples d'utilisation des fonctions

La fonction "absolu"

L'exemple suivant mémorise le résultat de l'appel à la fonction "absolu" dans une variable).

```
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    printf ("La valeur absolue de %d est : %d.\n ", x,y);
    return(0);
}
```

Exemples d'utilisation des fonctions

La fonction "absolu"

L'exemple suivant utilise l'appel à la fonction "absolu" dans une expression (qui calcule une distance dite de Manhattan).

```
int main (void)
{
    int absolu (int a);
    int x1, y1, x2, y2, distance;
    scanf("%d %d %d %d", x1, y1, x2, y2);
    distance=absolu(x1-y1)+absolu(x2-y2);
    return(0);
}
```

Passage
des paramètres

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
 if (a<0)
  return -a;
  else return a;
}
```

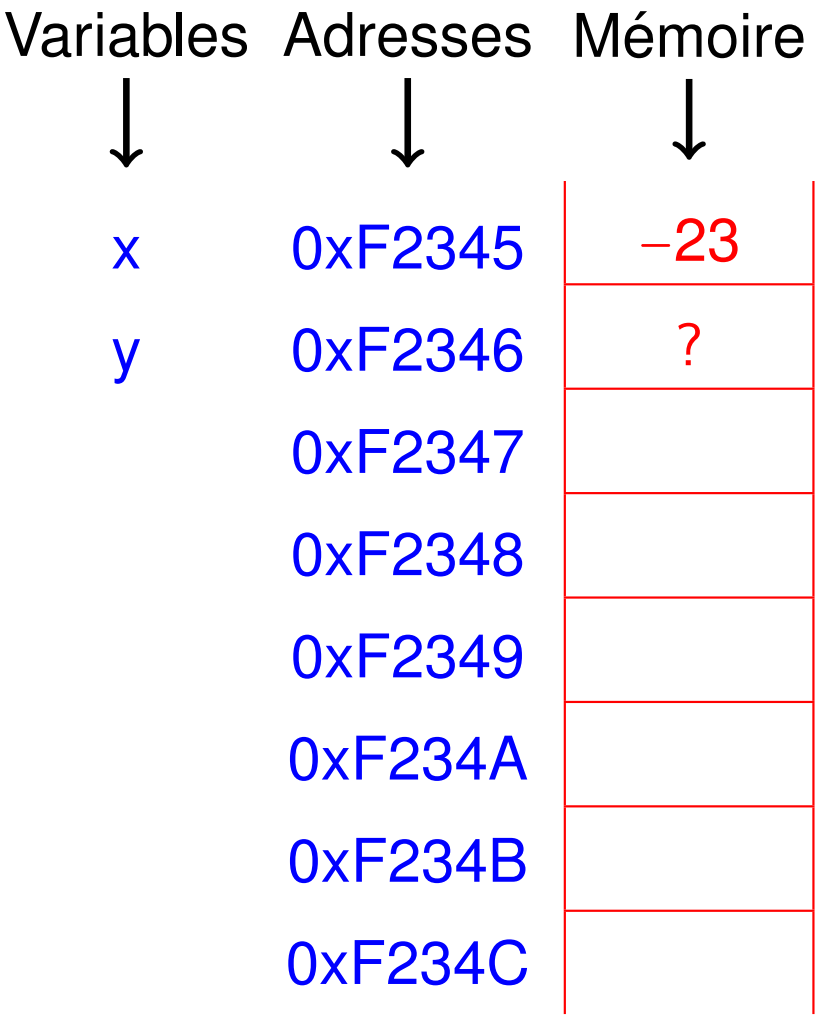
Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
 if (a<0)
  return -a;
  else return a;
}



int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
 if (a<0)
  return -a;
  else return a;
}
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

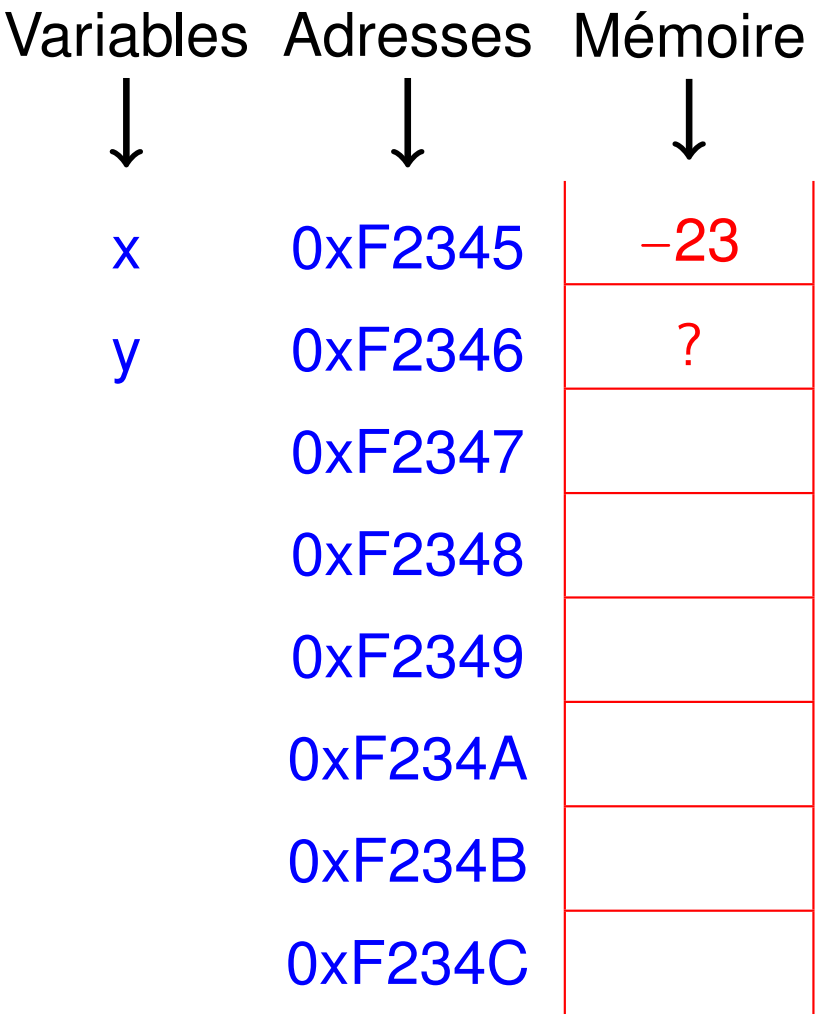
Voici donc l'état de la mémoire après les deux déclarations des variables x et y.



Regardons de près l'appel à la fonction "absolu"[2]

```
#include <stdio.h>
int absolu (int a)
{
 if (a<0)
  return -a;
  else return a;
}

int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

A ce niveau là, la fonction "absolu" est appelée et sera exécutée.

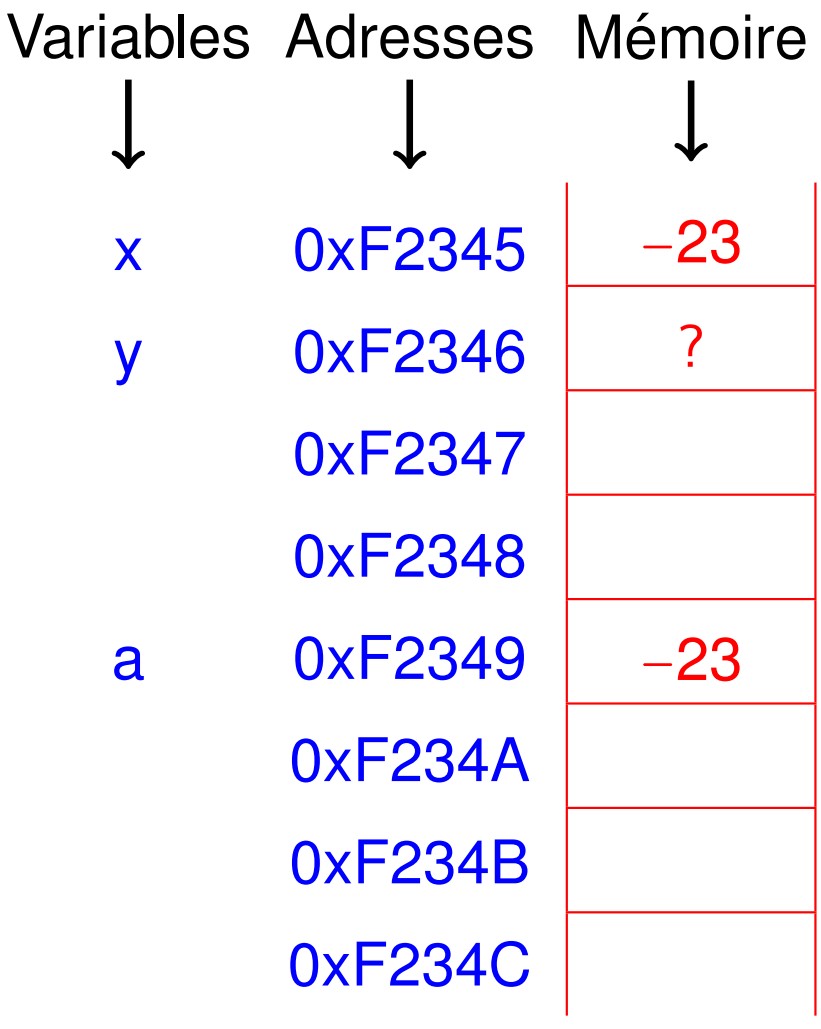


[2]Normalement, je dois laisser 4 octets minimum pour un entier

Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
 if (a<0)
  return -a;
  else return a;
}
int main (void)
{
     int absolu (int a);
     int x=-23, y;
     y=absolu(x);
     return(0);
}
```

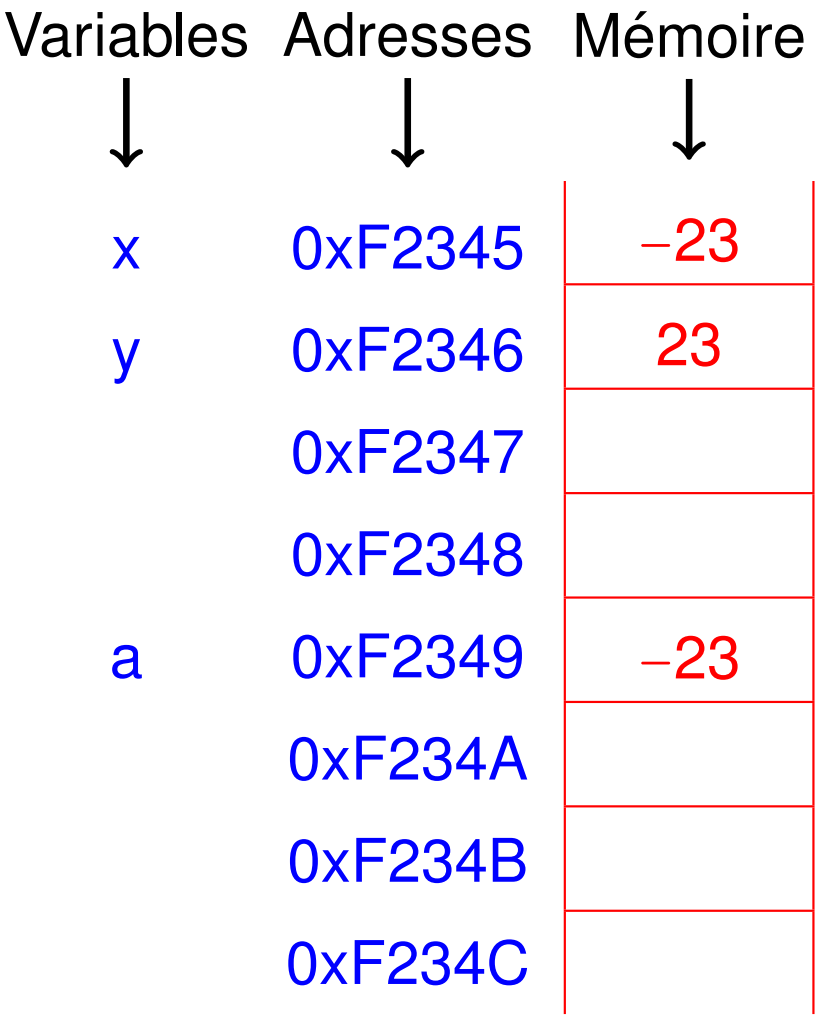
- Un espace mémoire est réservée pour *a*
- La valeur de x est recopiée dans *a*
- La valeur 23 est retournée



Regardons de près l'appel à la fonction "absolu"

```
#include <stdio.h>
int absolu (int a)
{
 if (a<0)
  return -a;
  else return a;
}
int main (void)
{
    int absolu (int a);
    int x=-23, y;
    y=absolu(x);
    return(0);
}
```

Finalement, la valeur (23) retournée par la fonction "absolu" est stockée dans la variable y.



Remarques

- Lors de l'appel :

$$y = absolu(x);$$

  Le compilateur créé :
  - un espace mémoire pour le paramètre *a*, et
  - recopie la valeur de *x* dans *a*
- Ce qui signifie que :
  - *x* et *a* sont stockées dans deux espaces mémoires différents,
  - et que toute modification apportée sur le paramètre *a* n'aura aucun effet sur la variable *x* déclarée dans la fonction "main".

Remarque importante sur le passage des paramètres

Le passage des paramètres dans les fonctions se fait par valeur.

Passage des paramètres : question

Question

Comment faire alors pour modifier le contenu d'une variable passée en paramètre?

Réponse

Utiliser dans ce cas l'adresse de la variable.

Passage par valeur : petit soucis

Ecrire une fonction C, appelée echange, qui :

- prend deux entiers positifs A et B en paramètres et les échange (en utilisant une variable intermédiaire).

Passage par valeur : petit soucis

Une première solution serait :

```
2 void echange (unsigned int a, unsigned int b)
3 {
4   unsigned int c;
5   c=a;
6   a=b;
7   b=c;
8 }
```

Passage par valeur : petit soucis

Une première solution serait :

```
#include <stdio.h>
int main (void)
{
  void echange (unsigned int a, unsigned int b);
  unsigned int x, y;
  x=5;
  y=10;
  echange(x, y);
  printf ("Après l'appel à la fonction echange
           x=%u et y=%u !!!!!!\n", x,y);
  return(0);
}
```

- Après exécution de ce programme,

Passage par valeur : petit soucis

Une première solution serait :

```
#include <stdio.h>
int main (void)
{
  void echange (unsigned int a, unsigned int b);
  unsigned int x, y;
  x=5;
  y=10;
  echange(x, y);
  printf ("Après l'appel à la fonction echange
           x=%u et y=%u !!!!!!\n", x,y);
  return(0);
}
```

- Après exécution de ce programme, on affiche :
  - et après l'appel à la fonction "echange",
  - on affiche x=5 et y=10 !!!!!!

Passage par valeur : petit soucis

Une première solution serait :

```
#include <stdio.h>
int main (void)
{
  void echange (unsigned int a, unsigned int b);
  unsigned int x, y;
  x=5;
  y=10;
  echange(x, y);
  printf ("Après l'appel à la fonction echange
           x=%u et y=%u !!!!!!\n", x,y);
  return(0);
}
```

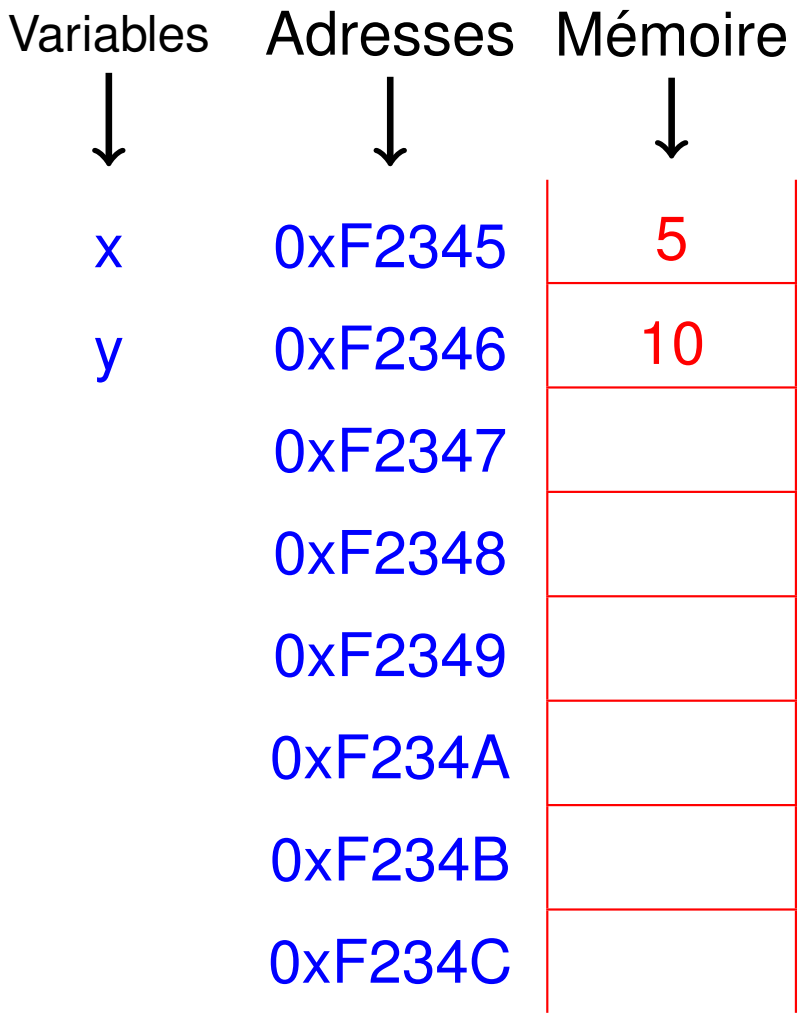
- Après exécution de ce programme, on affiche :
  - et après l'appel à la fonction "echange",
  - on affiche x=5 et y=10 !!!!!!
- C'est-dire les valeurs de x et y n'ont pas été échangées.

Fonction echange : Regardons le passage des paramètres en mémoire

Regardons de près l'appel à la fonction "echange" [3]

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;
 c=a;
 a=b;
 b=c;
}


int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;


 echange(x, y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

Voici l'état de la mémoire après l'exécution des deux premières instruction de la fonction main.

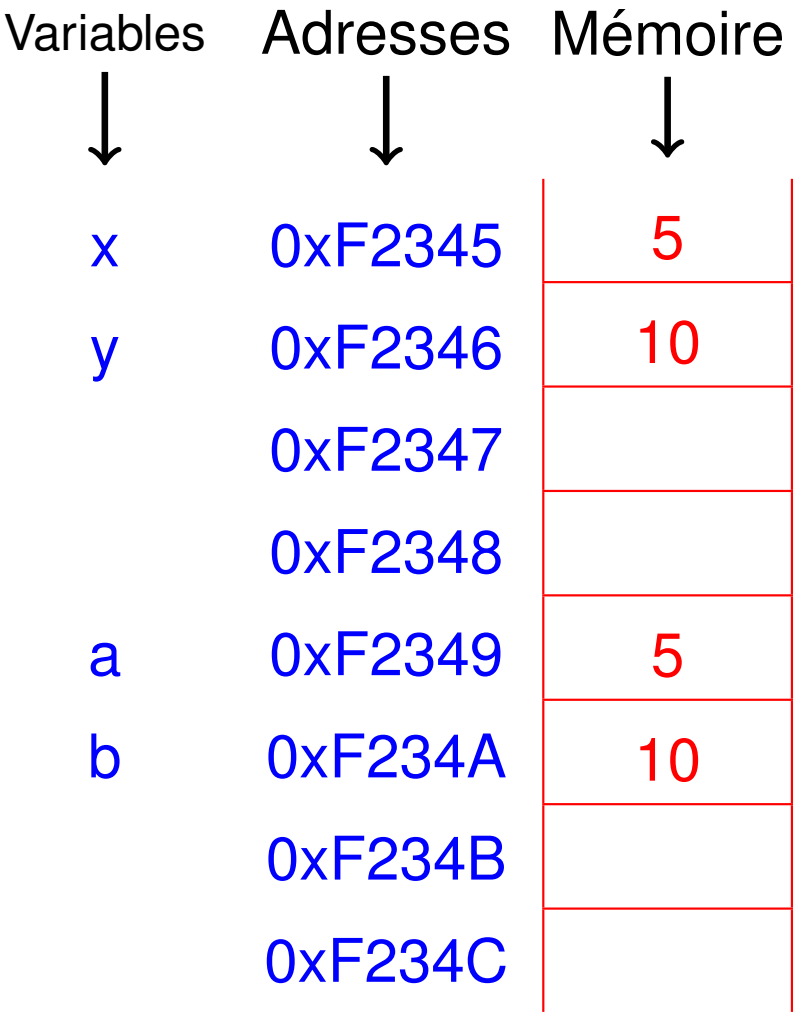


[3] Normalement, nous devons réserver 4 octets pour un entier.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;
 c=a;
 a=b;
 b=c;
}


int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(x, y);


 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

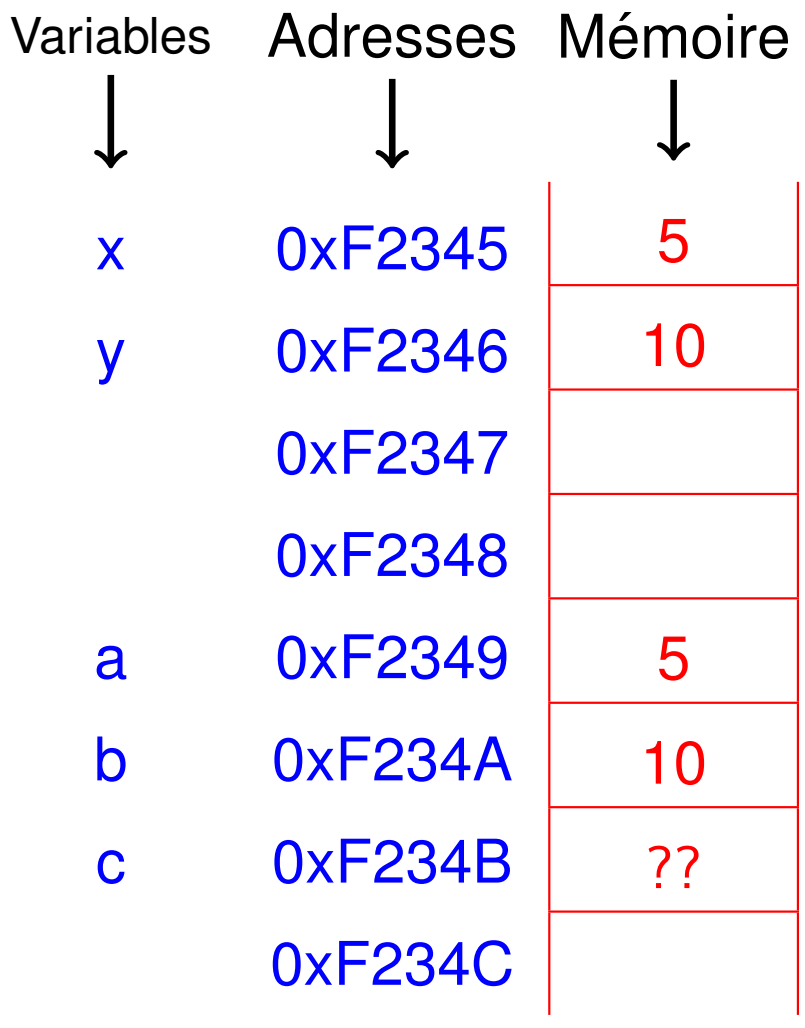
Etat de la mémoire au début de l'appel de la fonction échange et après passage des paramètres.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;

 c=a;
 a=b;
 b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(x, y);

 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

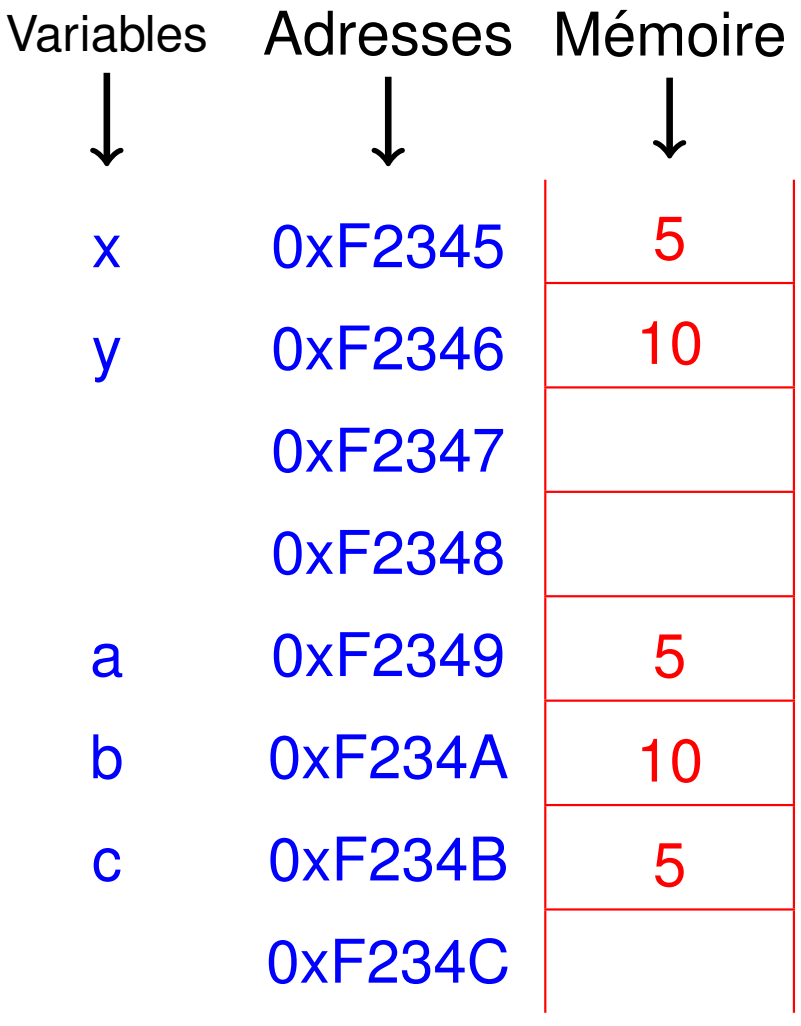
Etat de la mémoire après la déclaration de la variable locale c de la fonction échange.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;
 c=a;

 a=b;
 b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(x, y);

 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

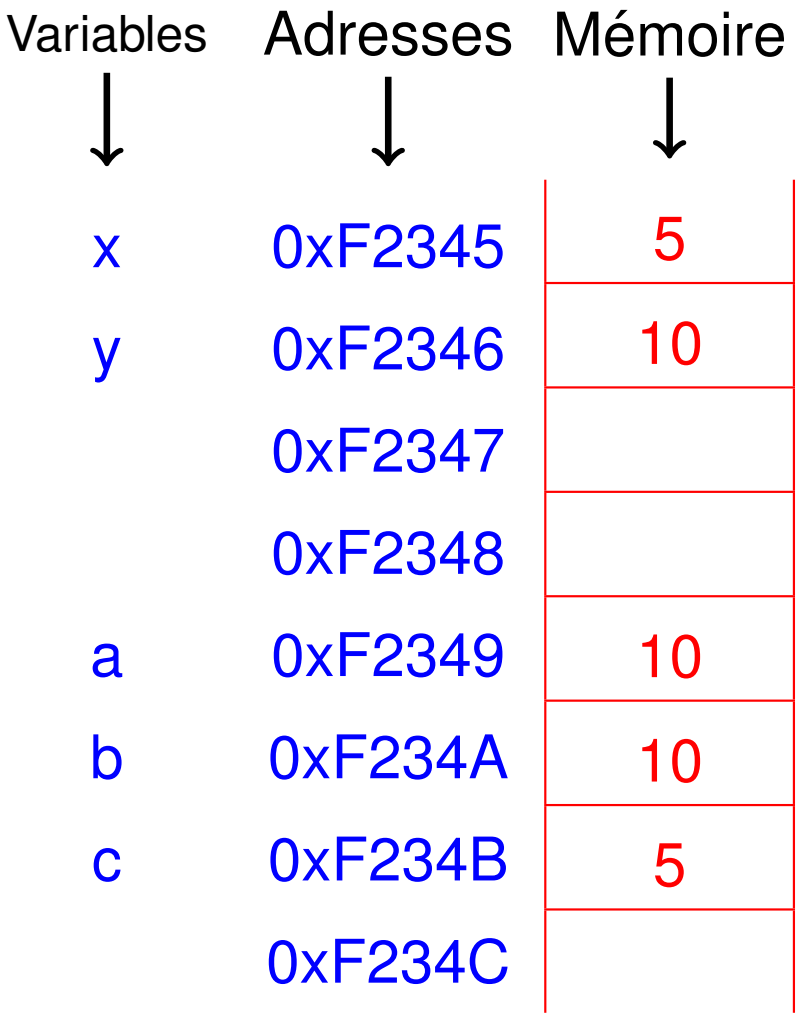
Etat de la mémoire après la première affectation de la fonction échange.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;
 c=a;
 a=b;
 b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(x, y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

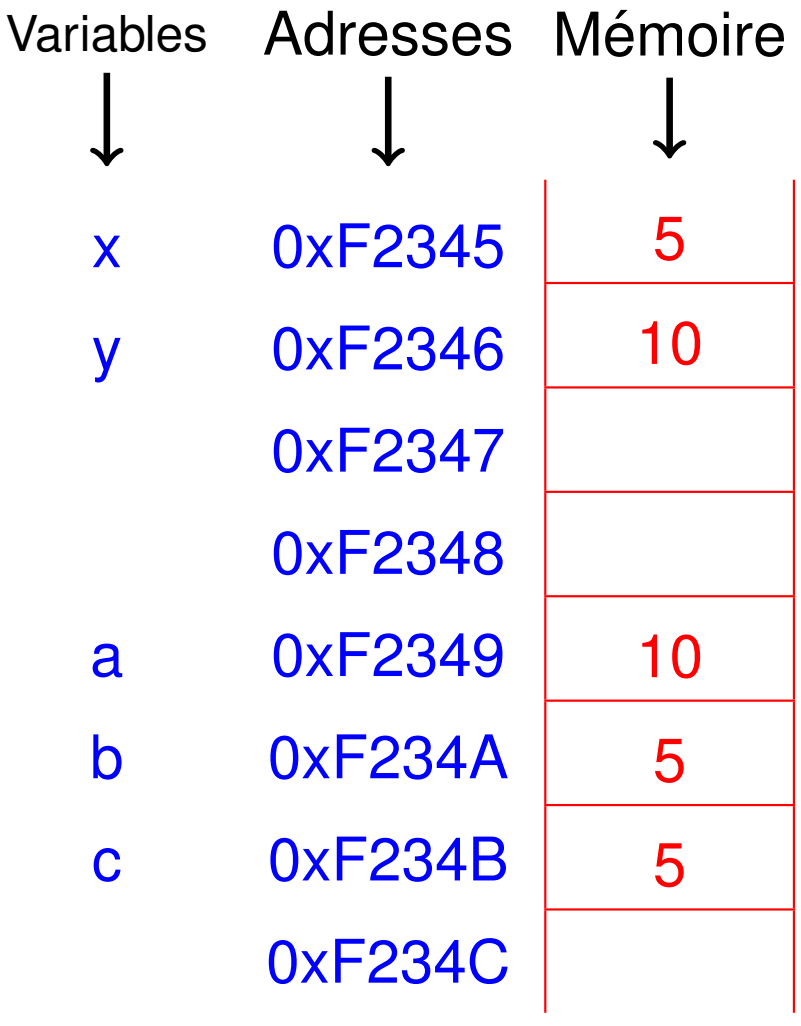
Etat de la mémoire après la deuxième affectation de la fonction échange.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;
 c=a;
 a=b;
 b=c;


}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(x, y);


 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

Etat de la mémoire après la dernière affectation de la fonction échange.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int a, unsigned int b)
{
 unsigned int c;
 c=a;
 a=b;
 b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;

 echange(x, y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

- Etat de la mémoire à la fin de l'appel de la fonction échange.
- Et malheureusement les valeurs n'ont pas été échangées.

Variables	Adresses	Mémoire
x	0xF2345	5
y	0xF2346	10
	0xF2347	
	0xF2348	
a	0xF2349	10
b	0xF234A	5
c	0xF234B	5
	0xF234C	

Fonction echange : passons les adresses des variables

Fonction echange : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echange" (depuis le main), de remplacer

```
echange(x, y);
```

  par

Fonction echange : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echange" (depuis le main), de remplacer

```
echange(x, y);
```

par

```
echange(&x, &y);
```

Fonction echange : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echange" (depuis le main), de remplacer

```
echange(x, y);
```

  par

```
echange(&x, &y);
```

- C'est-à-dire au lieu de passer la valeur des variables x et y, on passera les adresses où se trouvent les variables x et y.

Fonction echange : passons les adresses

Principe et première conséquence

- L'idée est lors de l'appel à la fonction "echange" (depuis le main), de remplacer

```
echange(x, y);
```

  par

```
echange(&x, &y);
```

- C'est-à-dire au lieu de passer la valeur des variables x et y, on passera les adresses où se trouvent les variables x et y.
- De cette façon, on peut accéder au contenu des variables x et y.

Fonction echange : passons les adresses

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction ”echange”.

Fonction echange : passons les adresses

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction ”echange”.
- Précisément, remplacer

```
1  void echange (unsigned int a, unsigned int b)
```

Fonction echange : passons les adresses

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction ”echange”.
- Précisément, remplacer

```
1 void echange (unsigned int a, unsigned int b)
```

par

```
1 void echange (unsigned int *a, unsigned int *b)
```

Fonction echange : passons les adresses

Deuxième conséquence

- Changer la déclaration des paramètres dans la fonction ”echange”.
- Précisément, remplacer

```
1 void echange (unsigned int a, unsigned int b)
```

par

```
1 void echange (unsigned int *a, unsigned int *b)
```

- C’est-à-dire, les paramètres *a* et *b* ne sont plus déclarés comme des entiers mais plutôt des pointeurs vers des entiers.

Fonction echange : passons les adresses

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction ”echange”.

Fonction echange : passons les adresses

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction ”echange”.
- Plus, précisément, remplacer

```
1    c=a;
2    a=b;
3    b=c;
```

Fonction echange : passons les adresses

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction ”echange”.
- Plus, précisément, remplacer

```
1    c=a;
2    a=b;
3    b=c;
```

par

```
1    c=*a;
2    *a=*b;
3    *b=c;
```

Fonction echange : passons les adresses

Troisième conséquence

- Changer les opérations arithmétiques dans la fonction ”echange”.
- Plus, précisément, remplacer

```
1    c=a;
2    a=b;
3    b=c;
```

par

```
1    c=*a;
2    *a=*b;
3    *b=c;
```

- C’est-à-dire, on a plus un accès direct aux données des variables ou des paramètres (ici a, b), mais seulement un accès indirect à ces données via les pointeurs.

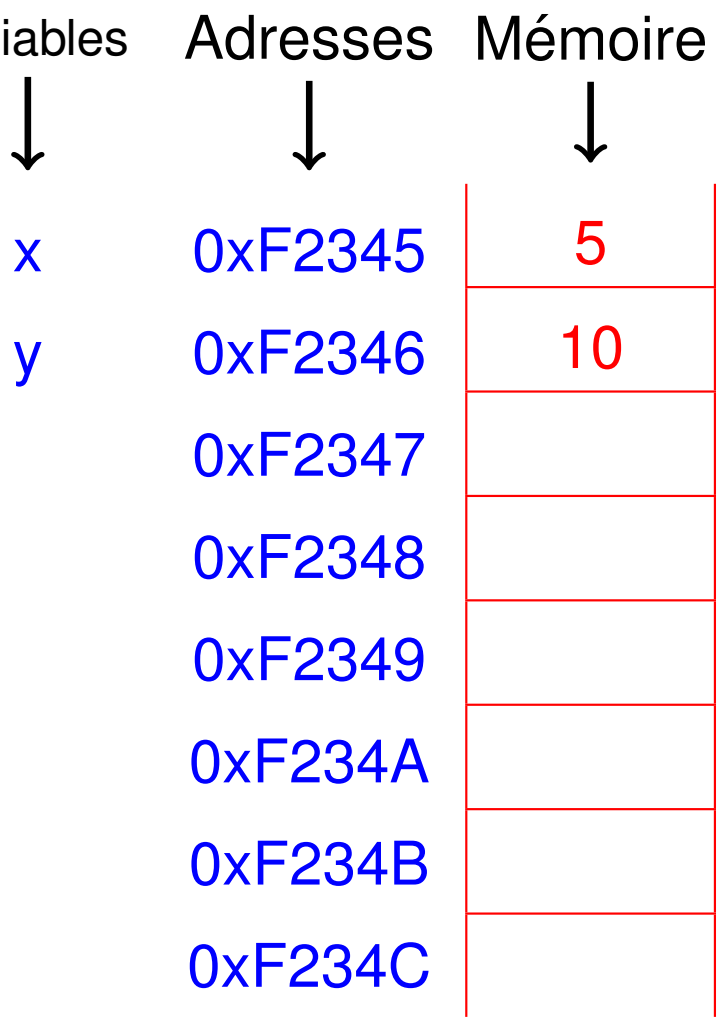
Fonction echange : passons les adresses

```
#include <stdio.h>
void echange (unsigned int *a, unsigned int *b)
{
  unsigned int c;
  c=*a;
  *a=*b;
  *b=c;
}

int main (void)
{
  void echange (unsigned int *a, unsigned int *b);
  unsigned int x, y;
  x=5;
  y=10;
  echange(&x, &y);
  printf ("Après l'appel à la fonction echange x=%u et y=%u.\n", x,y);
  return(0);
}
```

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;
 c=*a;
 *a=*b;
 *b=c;
}


int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;


 echange(&x, &y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

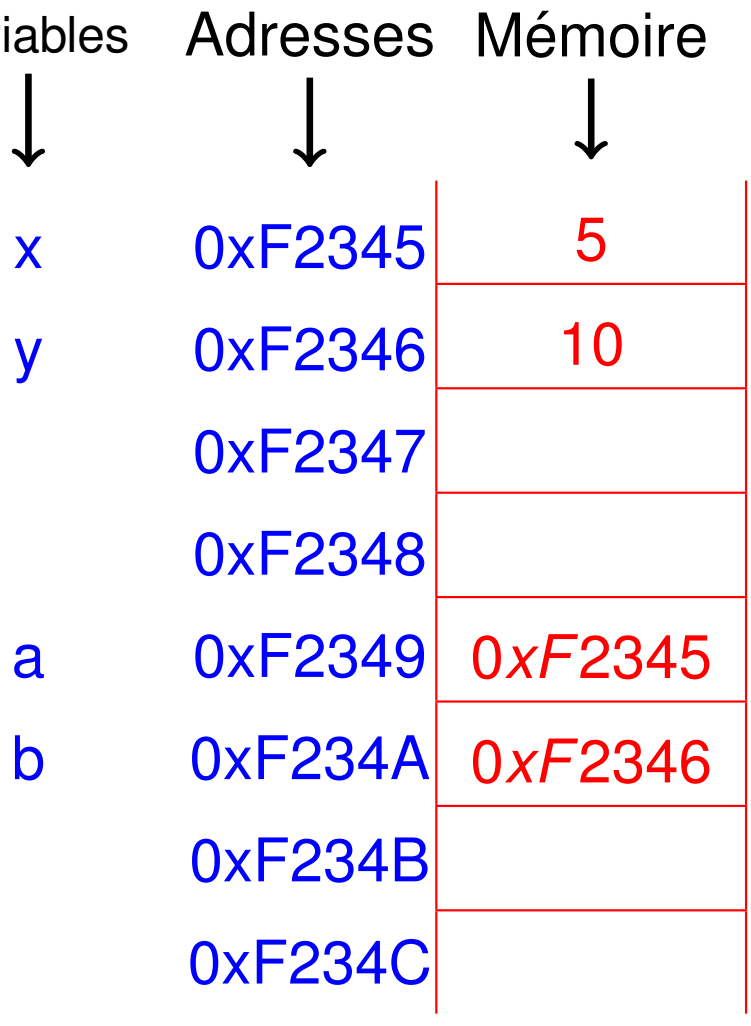
Voici l'état de la mémoire après l'exécution des deux premières instruction de la fonction main.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;
 c=*a;
 *a=*b;
 *b=c;
}


int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(&x, &y);


 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

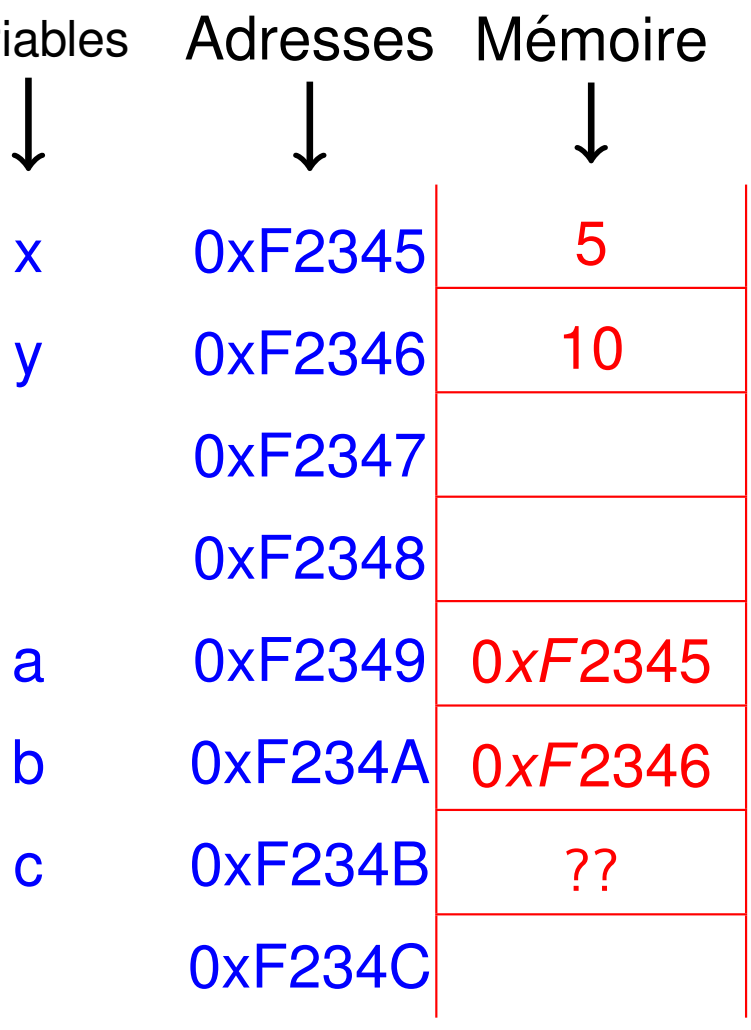
Etat de la mémoire au début de l'appel de la fonction échange et après passage des paramètres.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;

 c=*a;
 *a=*b;
 *b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(&x, &y);

 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

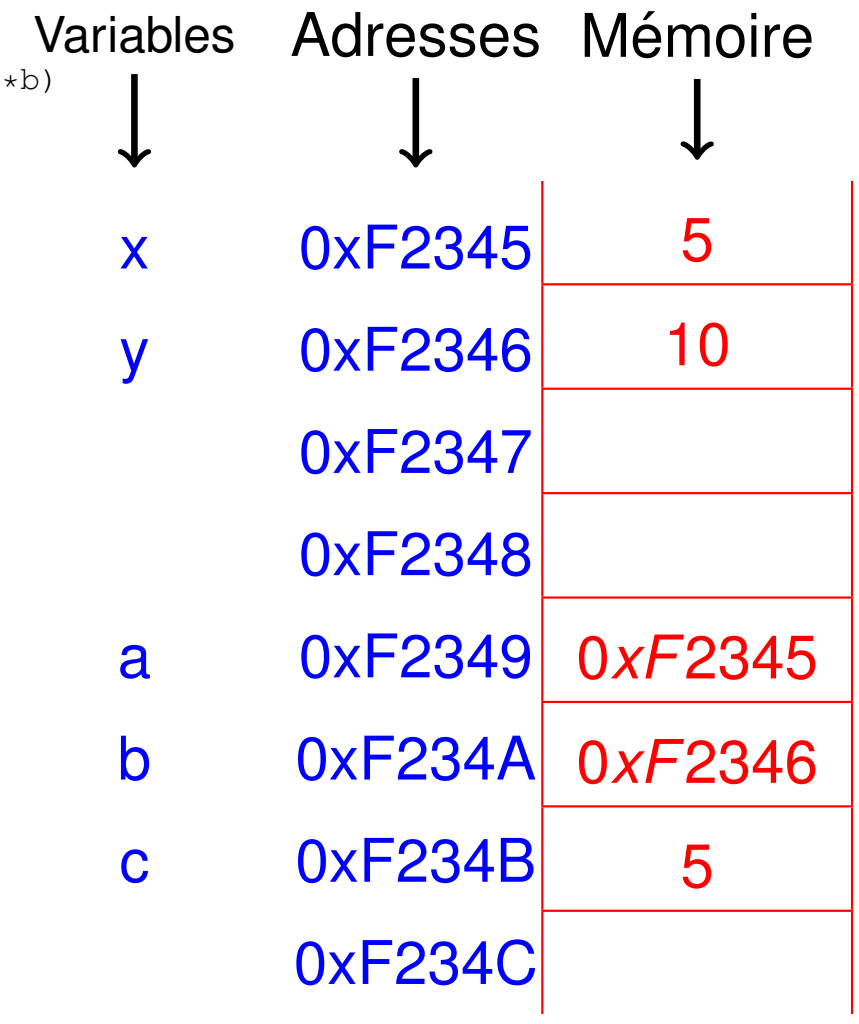
Etat de la mémoire après la déclaration de la variable locale c de la fonction échange.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;
 c=*a;
 *a=*b;
 *b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(&x, &y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

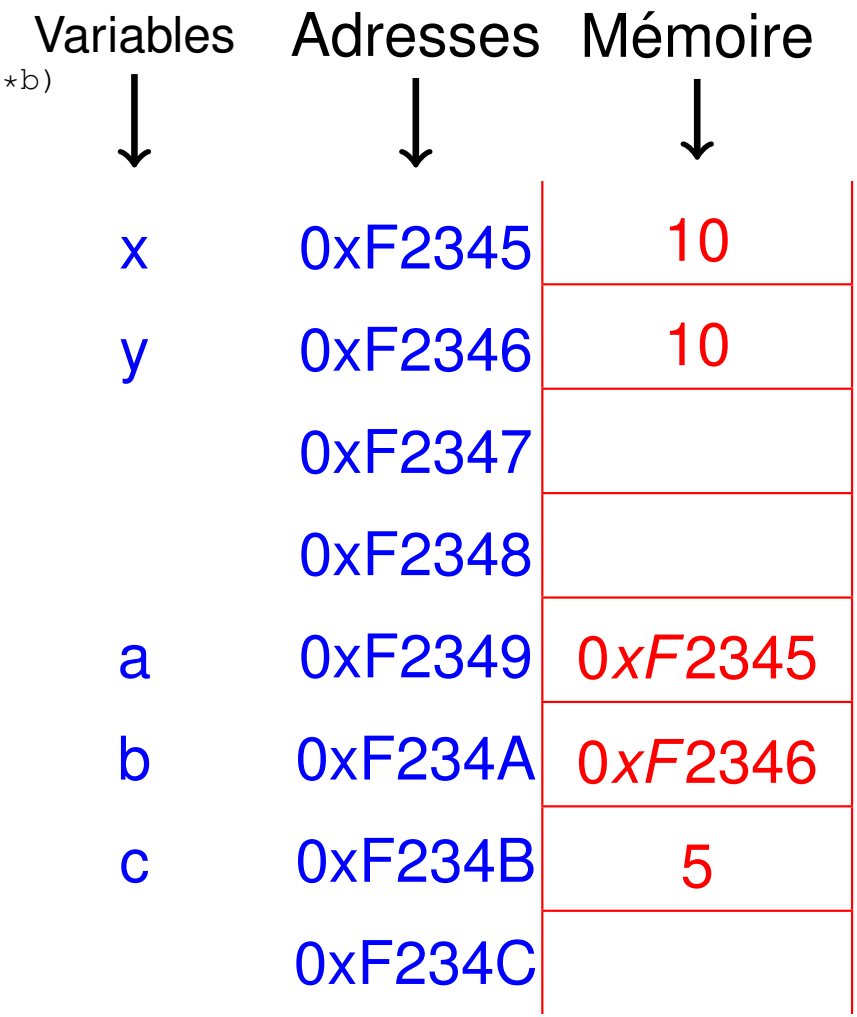
Etat de la mémoire après la première affectation de la fonction échange.



Regardons de près l'appel à la fonction "echange"

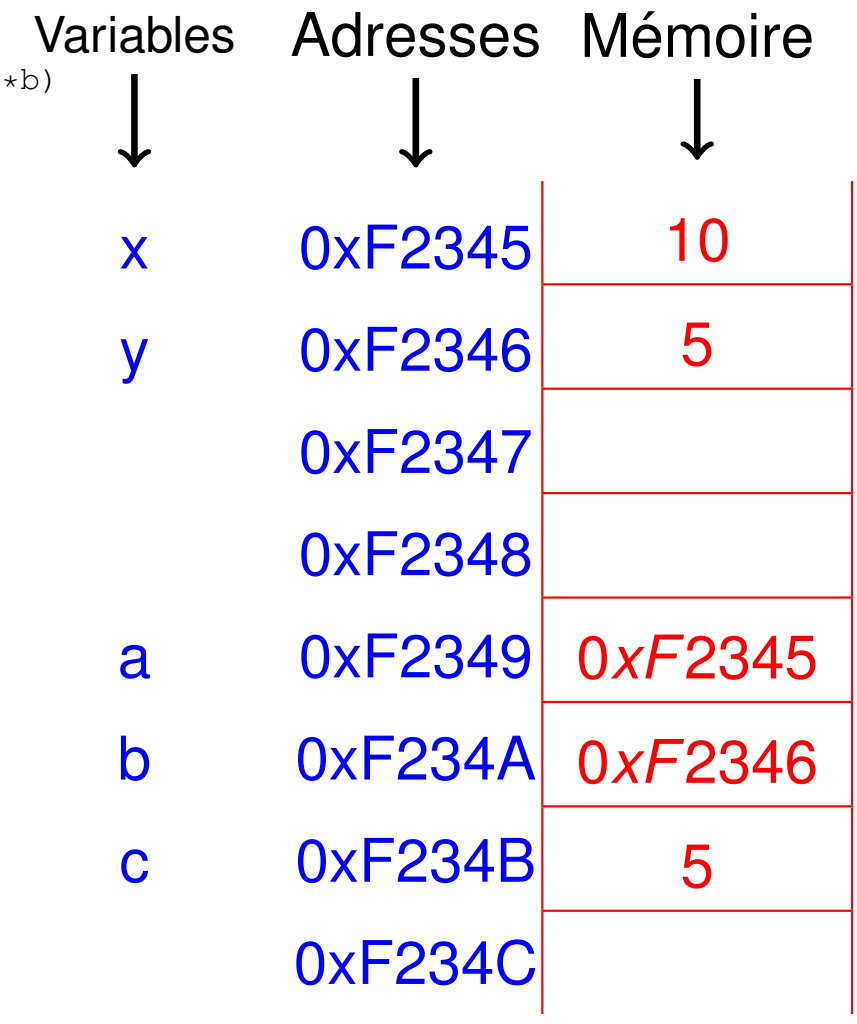
```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;
 c=*a;
 *a=*b;
 *b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(&x, &y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

Etat de la mémoire après la deuxième affectation de la fonction échange.



Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;
 c=*a;
 *a=*b;
 *b=c;


}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;
 echange(&x, &y);


 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

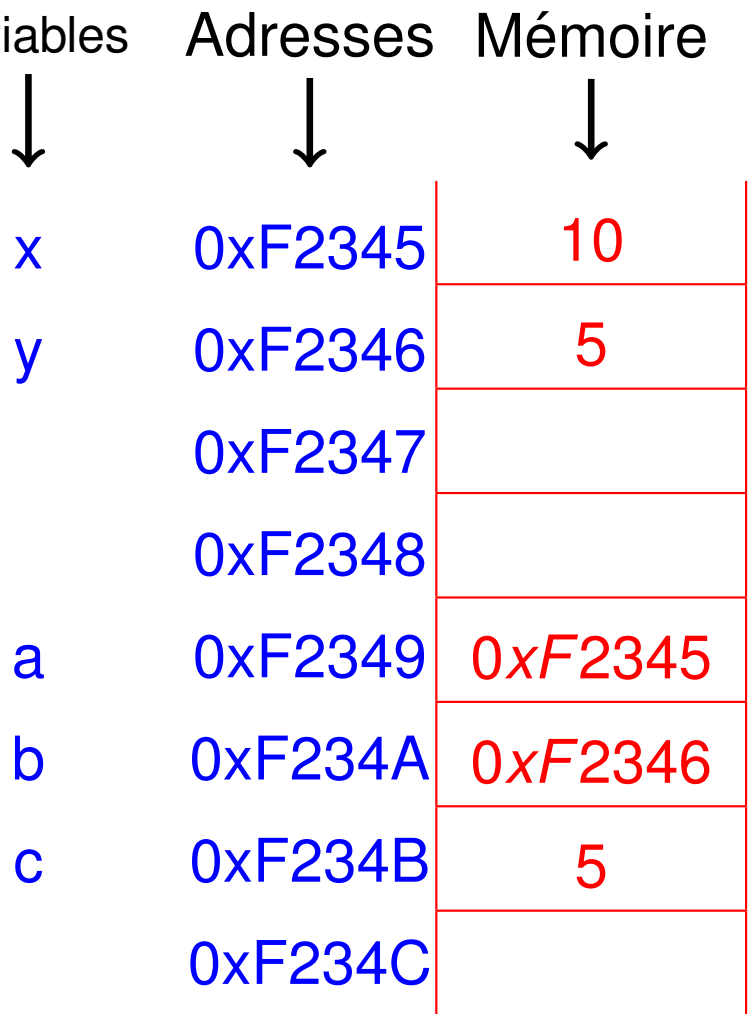


Etat de la mémoire après la dernière affectation de la fonction échange.

Regardons de près l'appel à la fonction "echange"

```
#include <stdio.h>
void echange(unsigned int *a, unsigned int *b)
{
 unsigned int c;
 c=*a;
 *a=*b;
 *b=c;
}
int main (void)
{
 unsigned int x, y;
 x=5;
 y=10;

 echange(&x, &y);
 printf ("Après l'appel à
la fonction echange x=%u et y=%u.\n", x,y);
 return(0);
}
```

- Etat de la mémoire à la fin de l'appel de la fonction échange.
- Et on affiche bien que les valeurs ont été échangées.



Notions
de
Variables locales

Variables locales

- Une variable est dite locale si elle a une portée (ou accessibilité) limitée.
- Toute variable déclarée à l'intérieur d'un bloc n'est accessible que dans ce bloc.
- Ceci est particulièrement vraie pour les fonctions :
  - La variable "c" n'est accessible qu'à l'intérieur de la fonction "echange".
  - Sa durée de vie = la durée de vie de la fonction.
  - De manière générale, dès qu'une fonction termine son exécution toutes les variables locales sont libérées et ne sont plus accessibles (sauf cas particuliers que nous discuterons plus tard).

Adresses et valeurs

Rappels : adresse des variables

Grâce à l'opérateur &, on peut récupérer les adresses des variables locales et paramètres utilisés dans la fonction Estpremiernaif.

```
2  int Estpremiernaif( int n)
3  {
4   int i;
5   if( n <= 1) return 0;
6   printf ("Dans la fonction Estpremiernaif :\n");
7   printf ("la variable n se trouve a l'adresse %p :\n", &n);
8   printf ("la variable local i se trouve a l'adresse %p :\n", &i);
9   for(i = 2; i <= n; ++i)
0   {
     if(n % i == 0) return 0;
2   }
3   return 1;
4  }
```

Adresses et valeurs

Adresse

- L'exécution de la fonction donne :

  Dans la fonction Estpremiernaif :

  la variable n se trouve a l'adresse : 0x7fff61f5cb2c
  la variable local i se trouve a l'adresse : 0x7fff61f5cb20

- Les adresses sont données en hexadécimal

Adresses et valeurs

Adresse des variables

Grâce à l'opérateur &, on peut également récupérer les adresses des fonctions.

```
2
3  int main ()
4  {
5  int nombre;
6  printf ("Introduire un nombre : ");
7  scanf ("%d", &nombre);
8  printf ("l\'adresse de la variable nombre est : %p \n", &nombre);
9  printf ("l\'adresse de la fonction Estpremiernaif est : %p \n",
0                 &Estpremiernaif);
   printf ("l\'adresse de la fonction main est : %p \n", &main);
2  if (Estpremiernaif(nombre)==1)
3   printf ("Le nombre %d est premier : ", nombre);
4   else printf ("Le nombre %d n est pas premier : ", nombre);
5  return(0);
6  }
```

Adresses et valeurs

L'adresse

- L'exécution de la fonction donne :

l'adresse de la variable nombre est :	0x7fff6d03db44
l'adresse de la fonction Estpremiernaif est :	0x10d43eba0
l'adresse de la fonction main est :	0x10d43ec50

Remarque

- La variable *nombre* utilisée dans le programme principal et le paramètre *n* de la fonction *Estpremiernaif* n'ont pas la même adresse.
- De ce fait, toute modification sur *n* n'affectera pas la variable *nombre*.

Les fonctions récursives

La Récursivité

Motivations

- La récursivité est essentielle dans de nombreux langages de programmation
- Elle permet d'offrir des solutions élégantes à de nombreux problèmes
- Elle s'applique particulièrement pour les problèmes qui se décomposent en sous-problèmes similaires mais de tailles plus petites. Elle reprend le principe de "diviser pour régner".

Motivations

Définition

Une fonction est dite récursive si elle utilise une référence à elle-même dans sa définition.

3 principes ...

3 principes ...

1. Une fonction récursive doit évidement avoir un appel récursif.
2. Une fonction doit contenir des cas de ”base” qui ne contiennent pas d’appels récursifs. Ca représente l’équivalent de conditions d’arrêt dans des procédure itératives. Sans ces situations de base, la fonction ”bouclera à l’infini”...
3. Les fonctions récursives doivent converger vers ces cas de base pour toutes les valeurs données en entrée.

Récursivité

Hypothèses de travail

- Etablir les cas de base (que l'on sait traiter facilement).
  - Ces cas de bases représentent les conditions d'arrêt où on ne fait plus d'appels récursifs
- Pour écrire une fonction récursive avec un paramètre donné (exemple un entier n), faites l'hypothèse que la fonction est disponible pour les paramètres plus petits (n-1, n-2, etc.)

Types de récursivité

Type de récursivité

Récursivité simple

Un algorithme est dit simplement récursif s'il ne fait qu'un seul appel à lui-même.

Exemple de récursivité simple

La factorielle de n

$$n! = n * (n-1)!,$$

et

$$0! = 1.$$

```
2   int factorielle(int n)
3   {
4     if (n==0) return 1;
5     else return factorielle(n-1)*n;
6   }
```

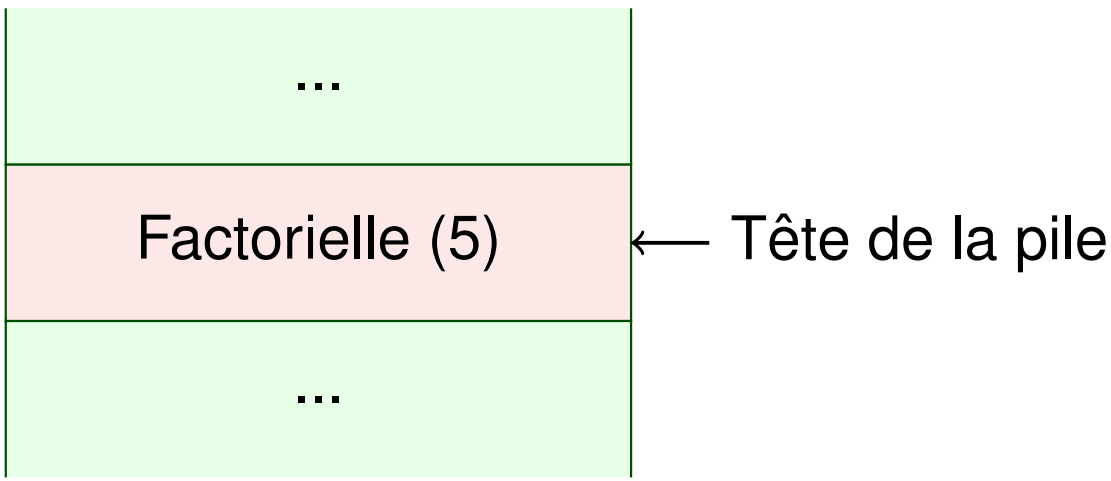
Exécution de la fonction factiorelle

Pile d'execution

La Pile d'exécution (stack) du programme en cours est un emplacement mémoire destiné à mémoriser les paramètres, les variables locales ainsi que les adresses de retour des fonctions en cours d'exécution.

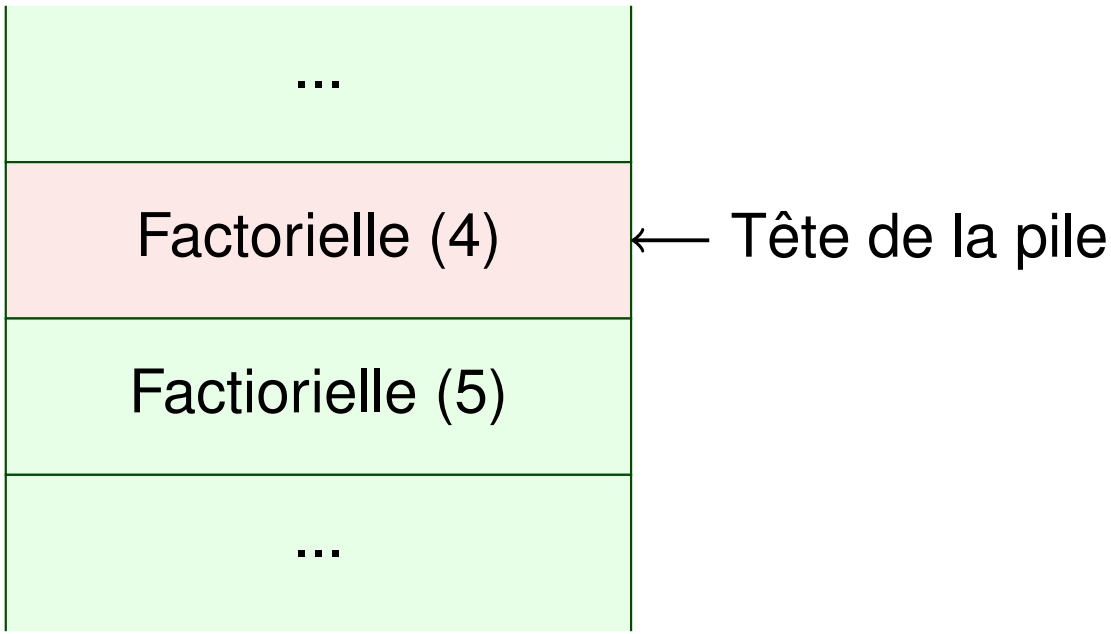
Appel à la fonction récursive : Etape = 1

On empile la fonction factorielle de 5



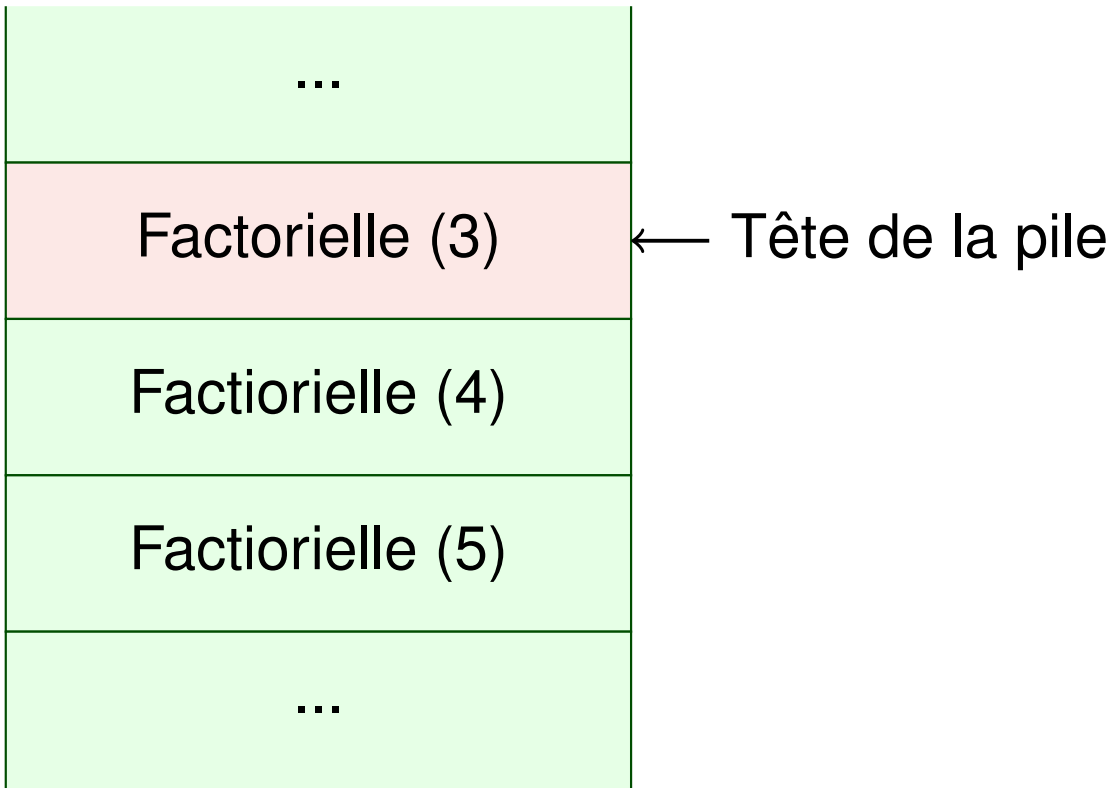
Appel à la fonction récursive : Etape = 2

On empile la fonction factorielle de 4



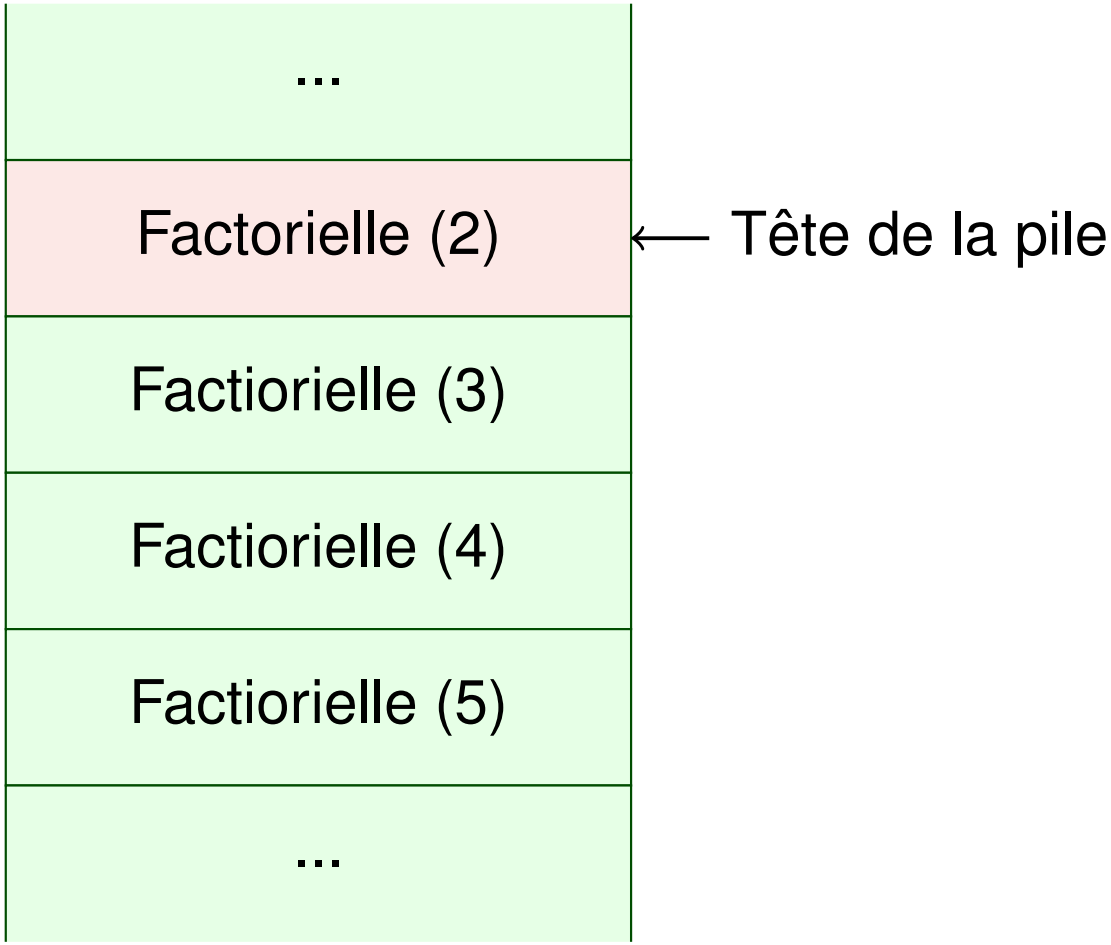
Appel à la fonction récursive : Etape = 3

On empile la fonction factorielle de 3



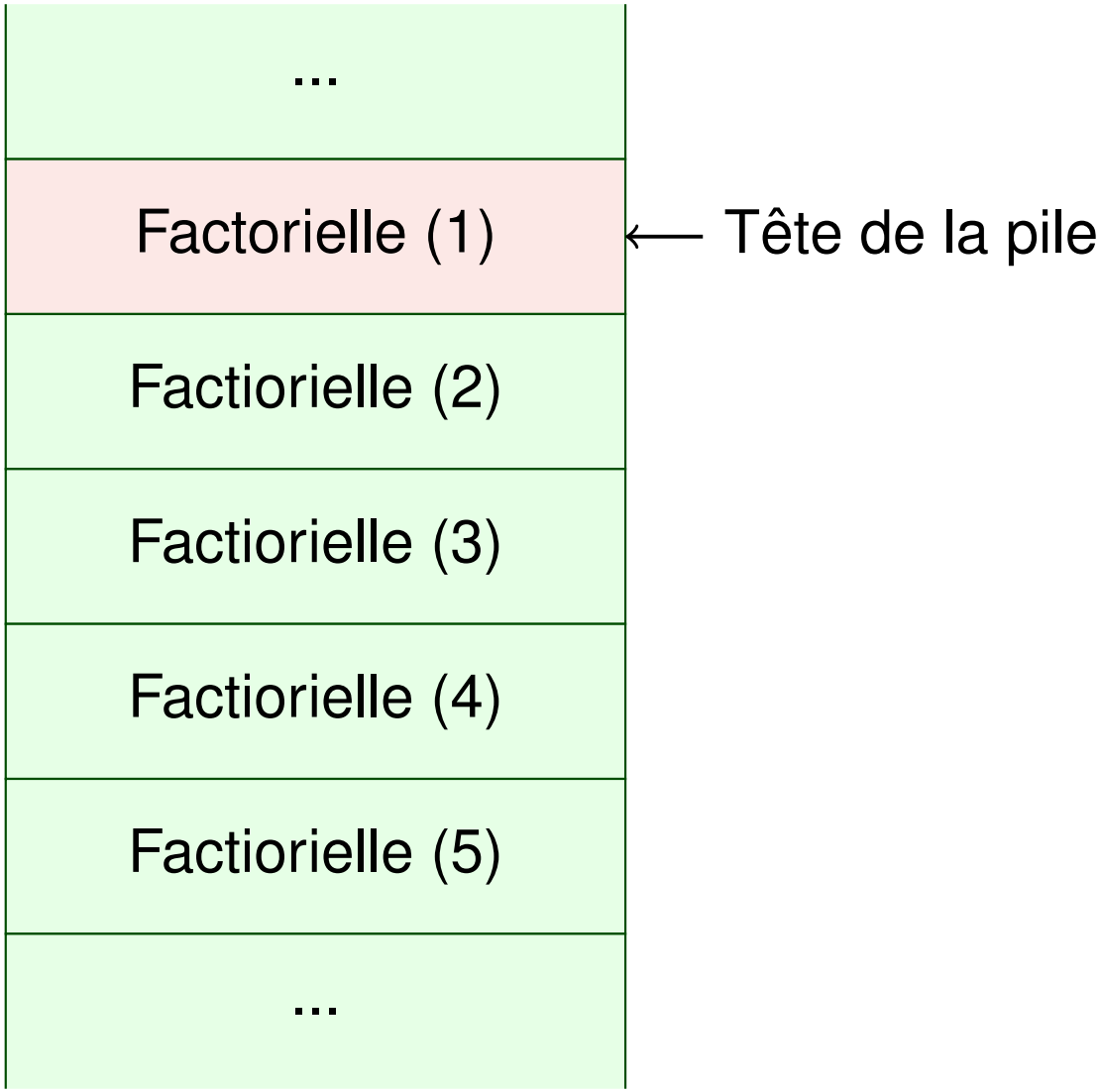
Appel à la fonction récursive : Etape = 4

On empile la fonction factorielle de 2



Appel à la fonction récursive : Etape = 5

On empile la fonction factorielle de 1



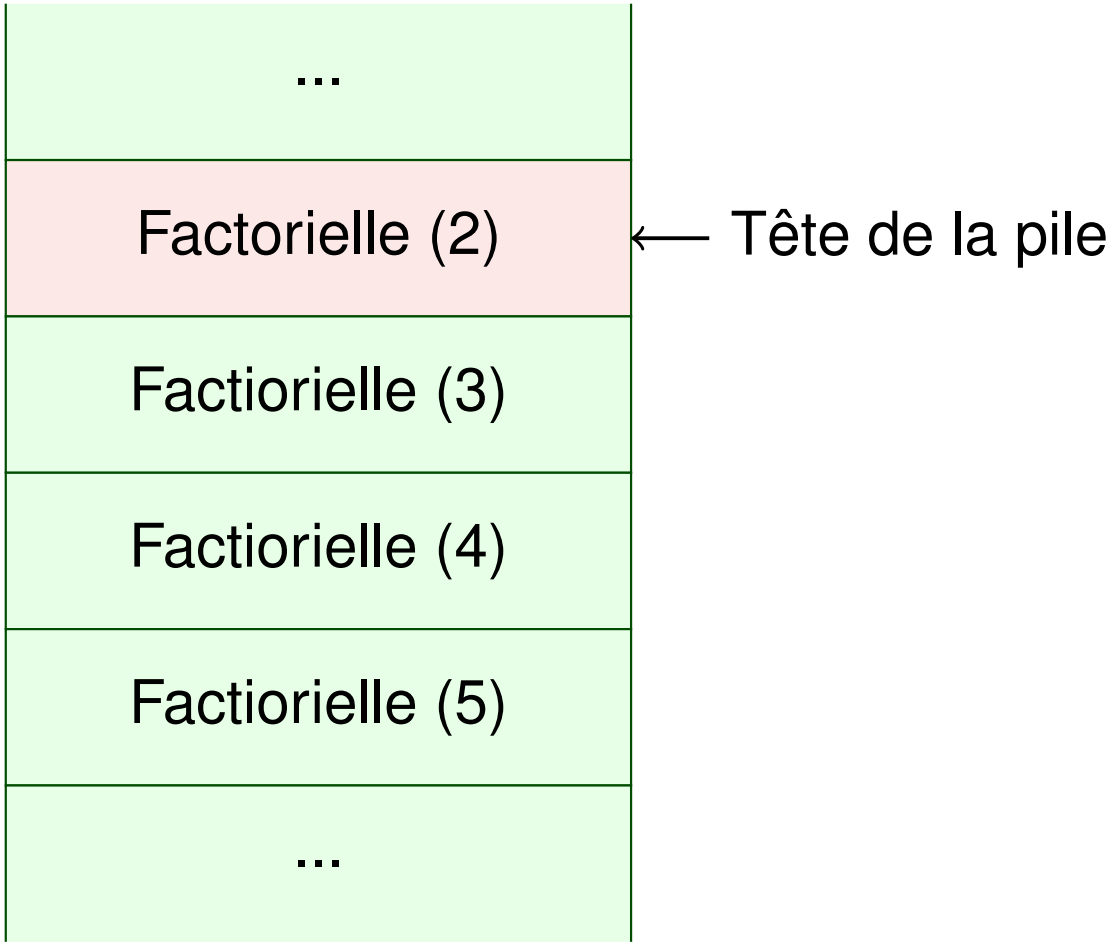
Résultat de la fonction récursive : Factiorelle(1)

On calcule la fonction factorielle de 1

$$\text{Factiorelle}(1) = 1 * \text{Factiorelle}(0) = 1.$$

Retour de la fonction récursive

On depile la fonction factorielle (1)



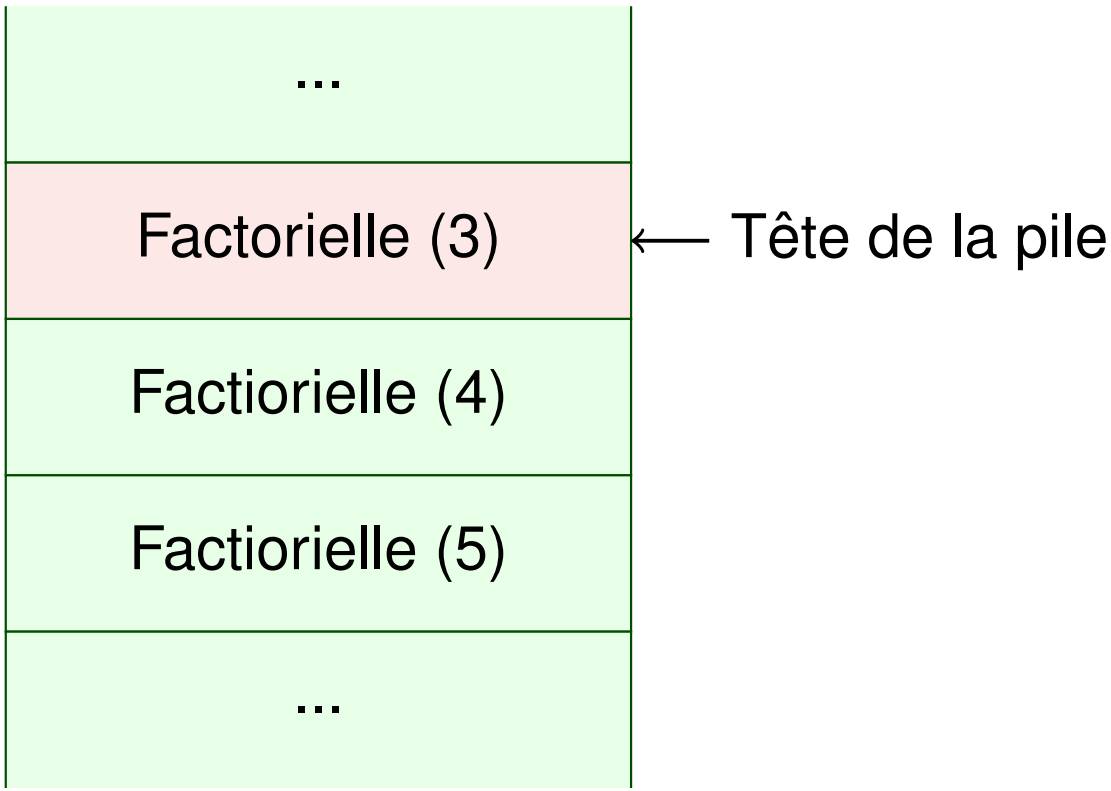
Résultat de la fonction récursive : Factiorelle(2)

On calcule la fonction factorielle de 2

$$\text{Factiorelle}(2) = 2 * \text{Factiorelle}(1) = 2.$$

Retour de la fonction récursive

On depile la fonction factorielle (2)



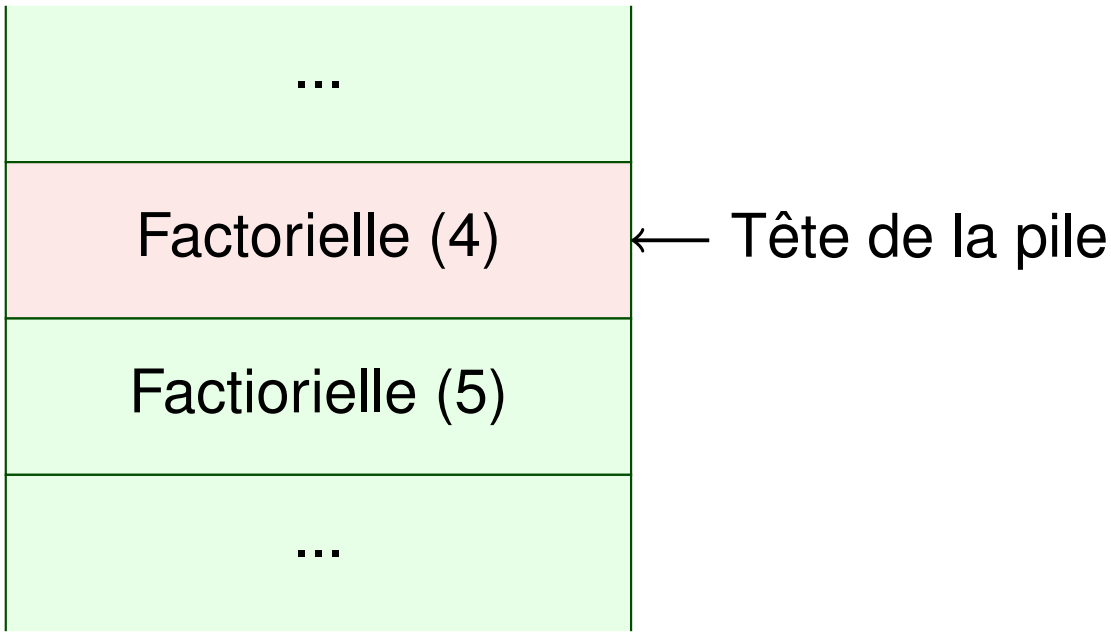
Résultat de la fonction récursive : Factiorelle(3)

On calcule la fonction factorielle de 3

$$\text{Factiorelle}(3) = 3 * \text{Factiorelle}(2) = 6.$$

Retour de la fonction récursive

On depile la fonction factorielle (3)



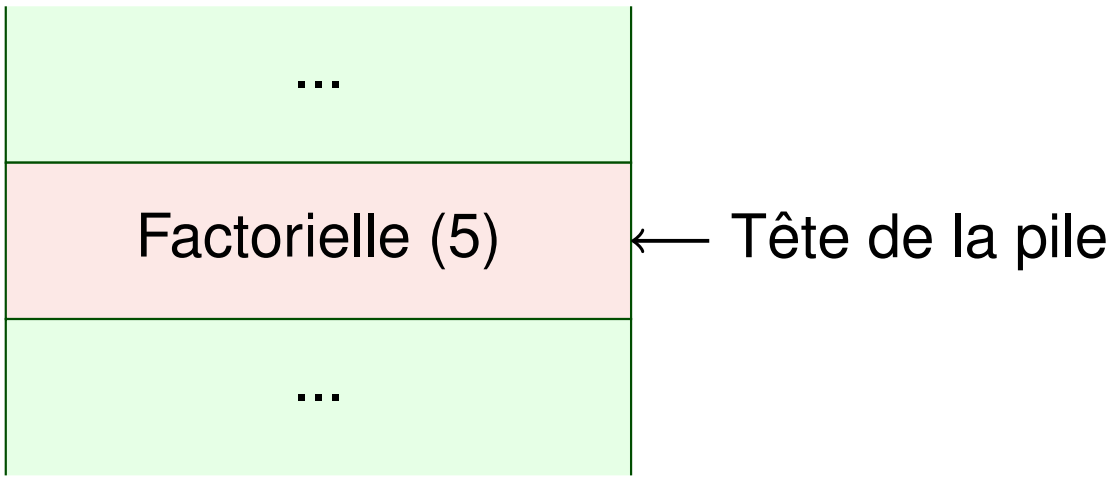
Résultat de la fonction récursive : Factiorelle(4)

On calcule la fonction factorielle de 4

$$\text{Factiorelle}(4) = 4 * \text{Factiorelle}(3) = 24.$$

Retour de la fonction récursive

On depile la fonction factorielle (4)



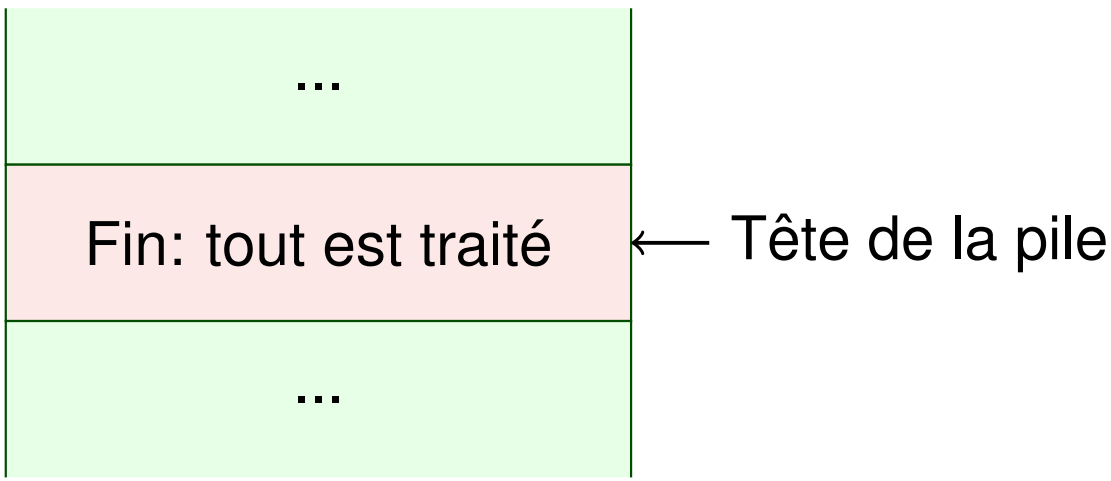
Résultat de la fonction récursive : Factiorelle(5)

On calcule la fonction factorielle de 5

$$\text{Factiorelle}(5) = 5 * \text{Factiorelle}(4) = 120.$$

Retour de la fonction récursive

On depile la fonction factorielle (5)



Tour de Hanoï

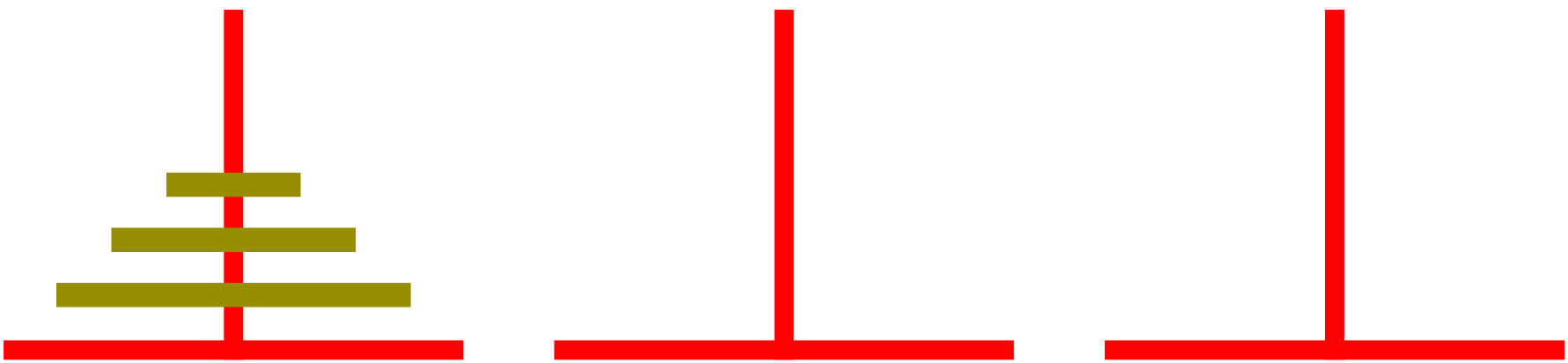
et ses variantes

Tour de Hanoï :

nombre de disques = 3

Tour de Hanoï

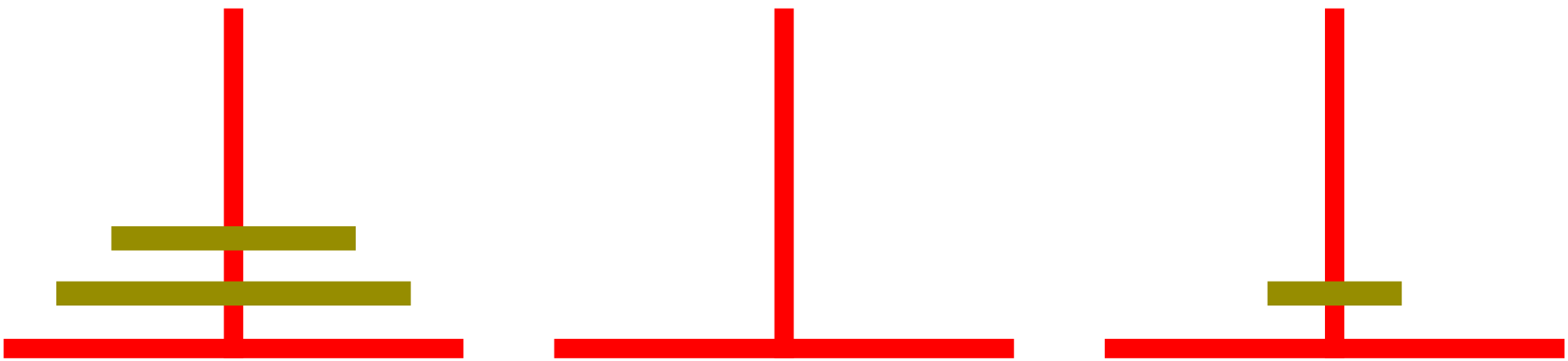
Etape 0: La tour à l'état initial



Tige 0 Tige 1 Tige 2

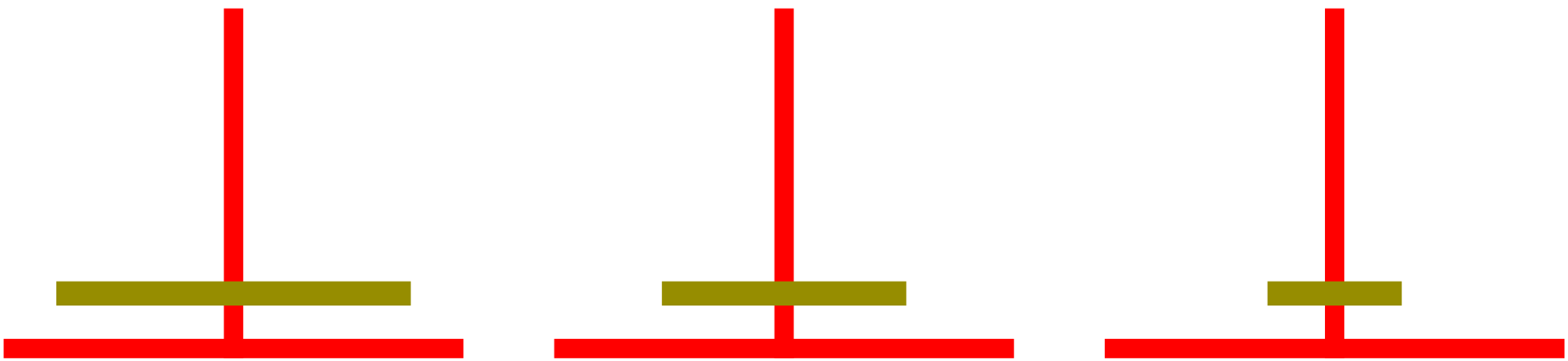
Tour de Hanoï

Etape 1: déplacer un disque de la tige 0 vers la tige 2



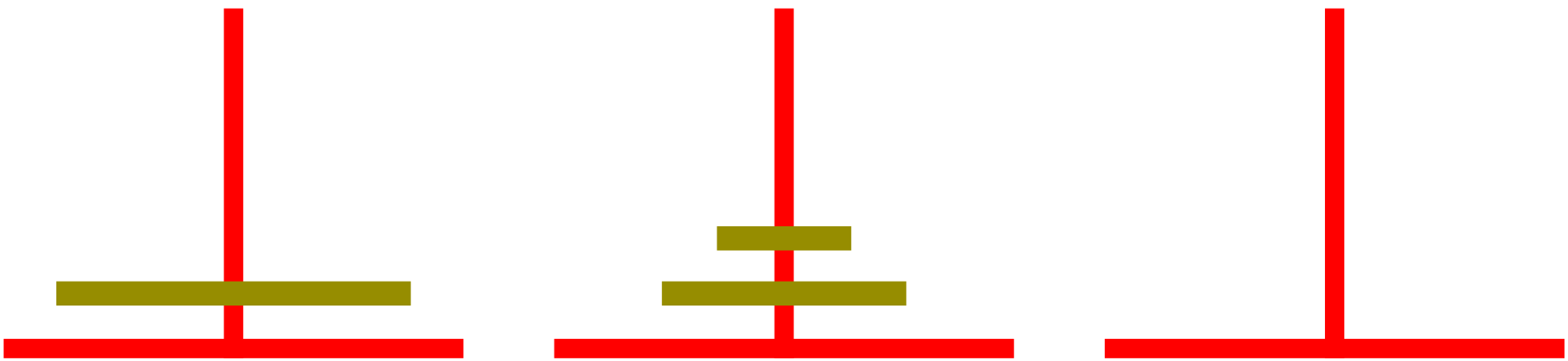
Tour de Hanoï

Etape 2: déplacer un disque de la tige 0 vers la tige 1



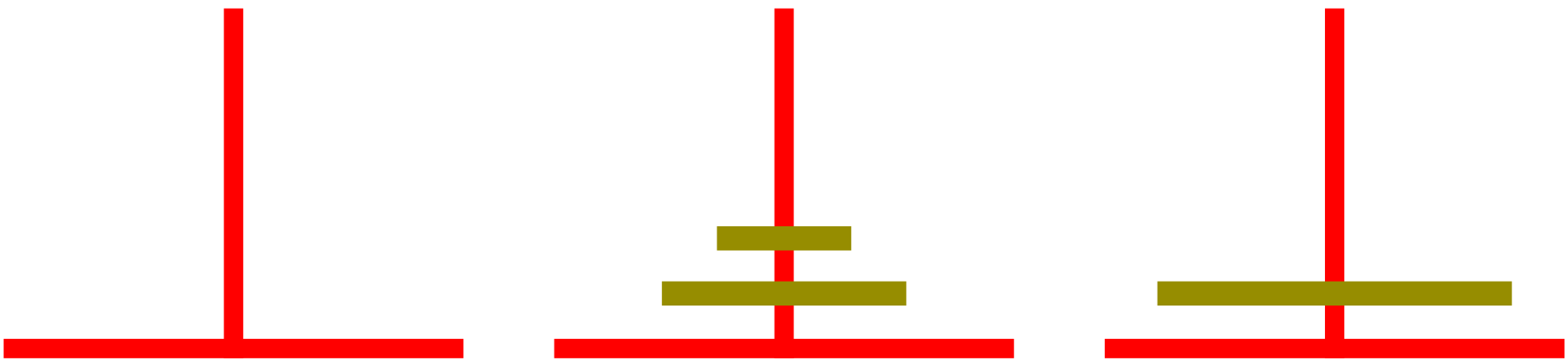
Tour de Hanoï

Etape 3: déplacer un disque de la tige 2 vers la tige 1



Tour de Hanoï

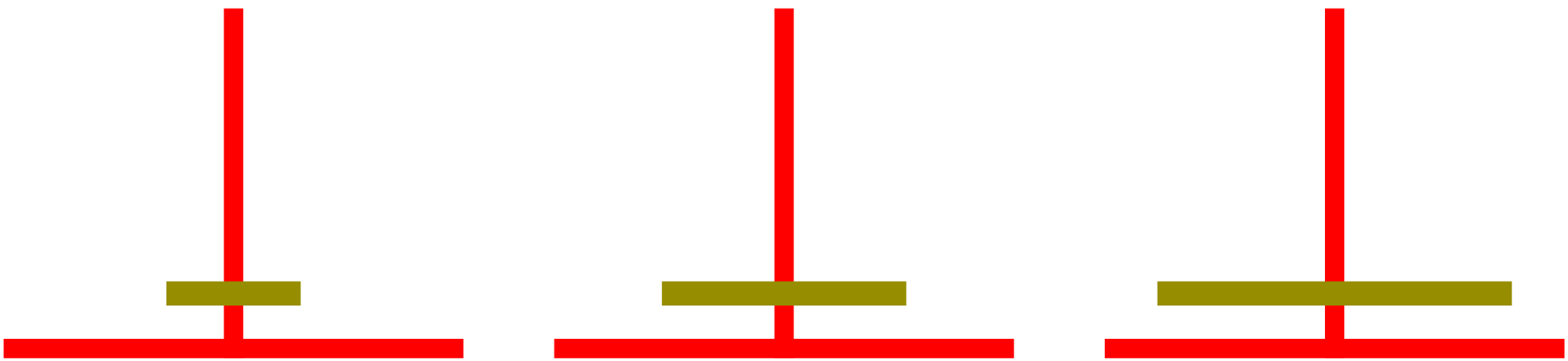
Etape 4: déplacer un disque de la tige 0 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

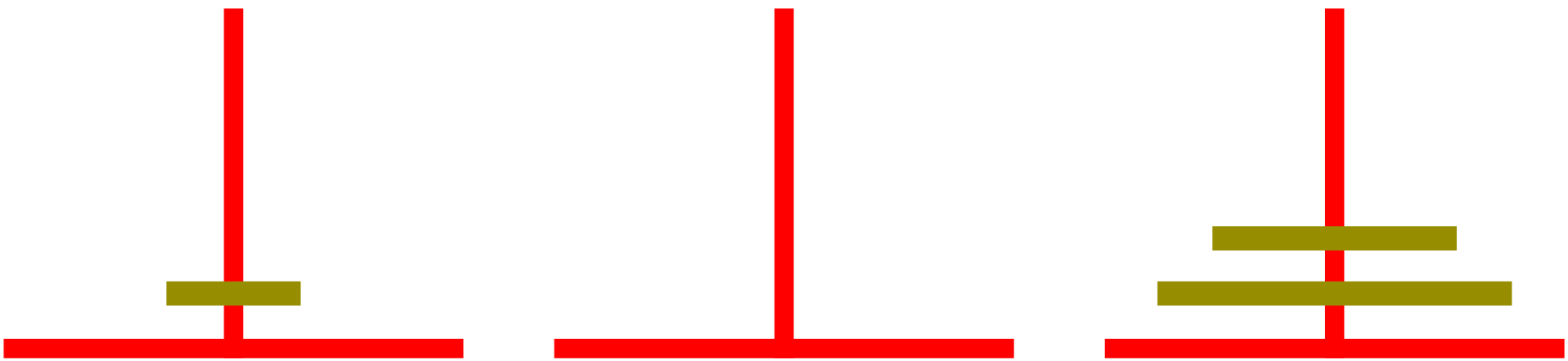
Etape 5: déplacer un disque de la tige 1 vers la tige 0



Tige 0 Tige 1 Tige 2

Tour de Hanoï

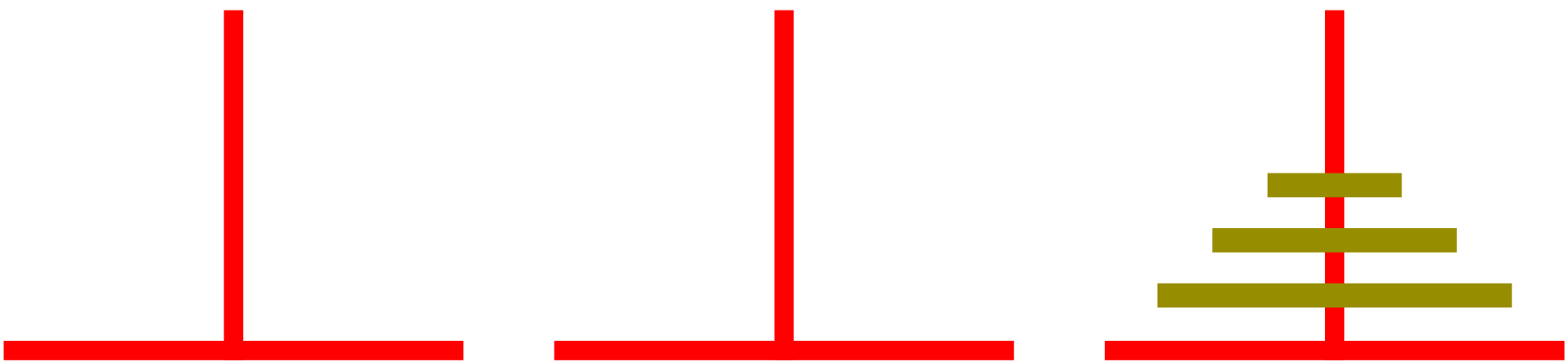
Etape 6: déplacer un disque de la tige 1 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

Etape 7: déplacer un disque de la tige 0 vers la tige 2



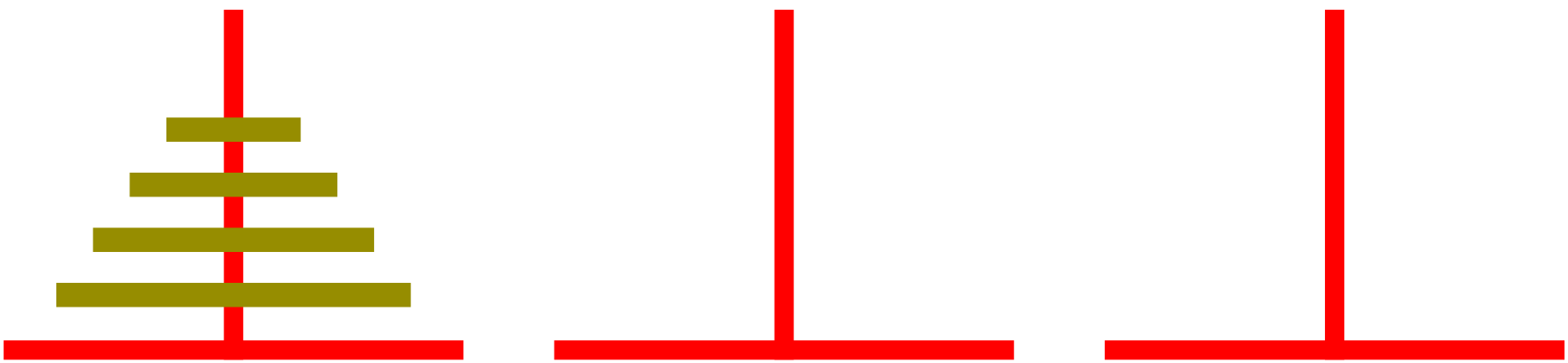
Tige 0 Tige 1 Tige 2

Tour de Hanoï :

nombre de disques = 4

Tour de Hanoï

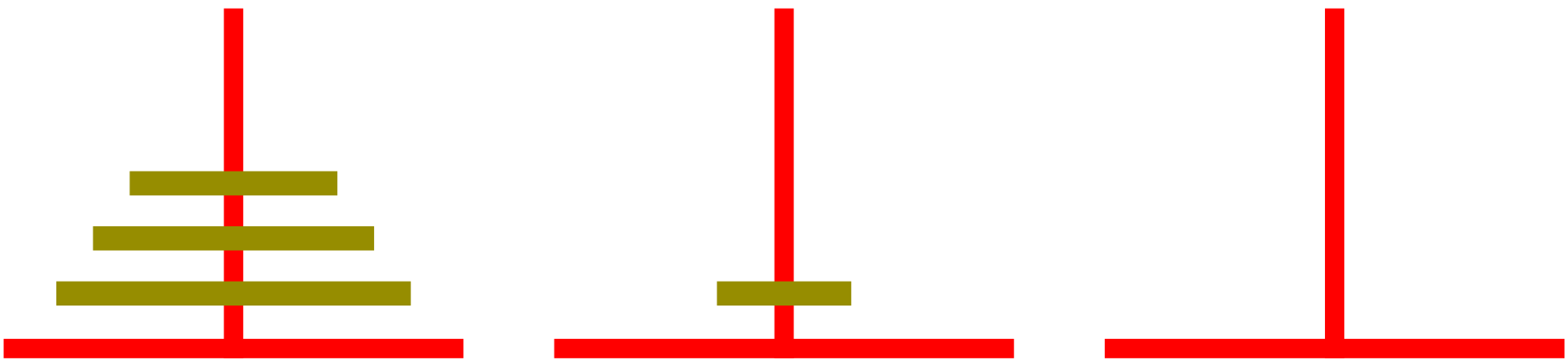
Etape 0: La tour à l'état initial



Tige 0 Tige 1 Tige 2

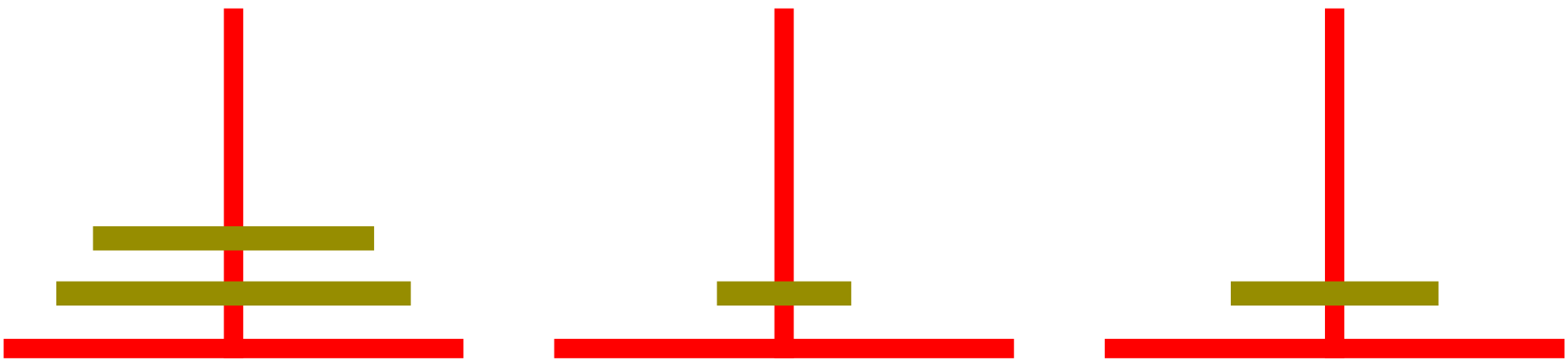
Tour de Hanoï

Etape 1: déplacer un disque de la tige 0 vers la tige 1



Tour de Hanoï

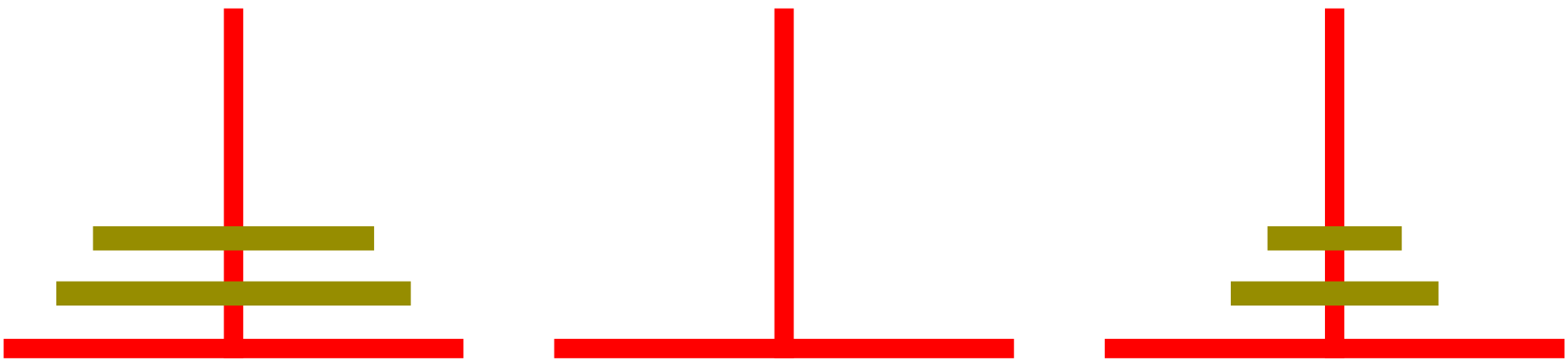
Etape 2: déplacer un disque de la tige 0 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

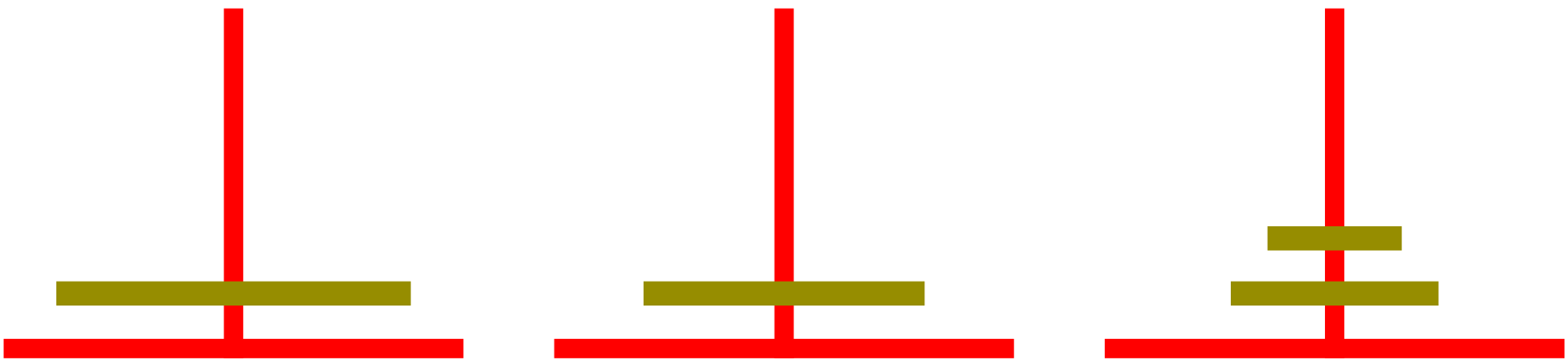
Etape 3: déplacer un disque de la tige 1 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

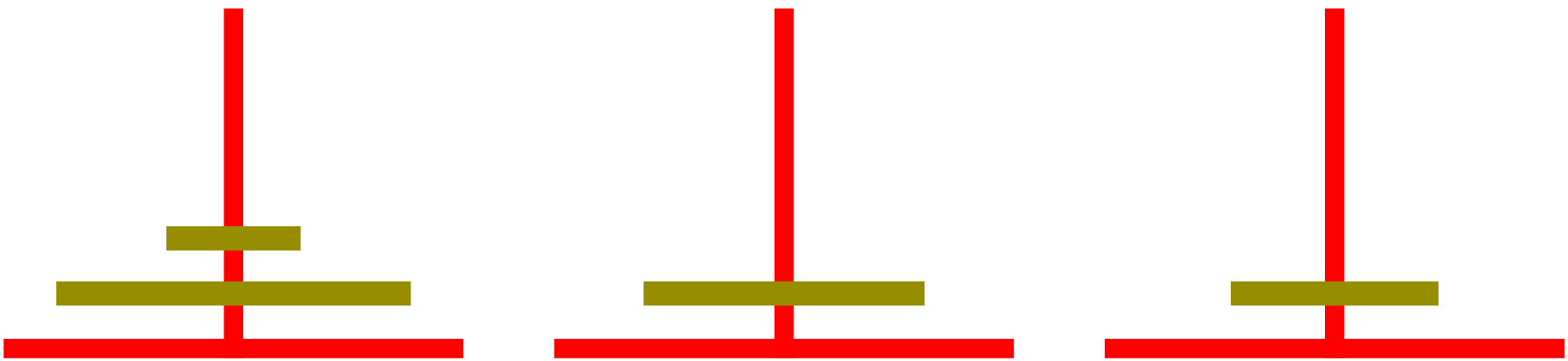
Etape 4: déplacer un disque de la tige 0 vers la tige 1



Tige 0 Tige 1 Tige 2

Tour de Hanoï

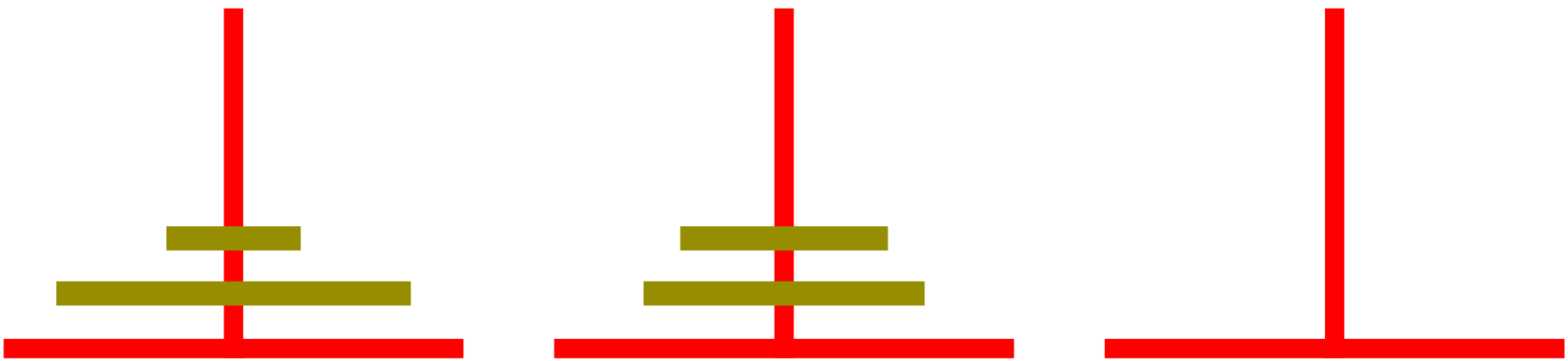
Etape 5: déplacer un disque de la tige 2 vers la tige 0



Tige 0 Tige 1 Tige 2

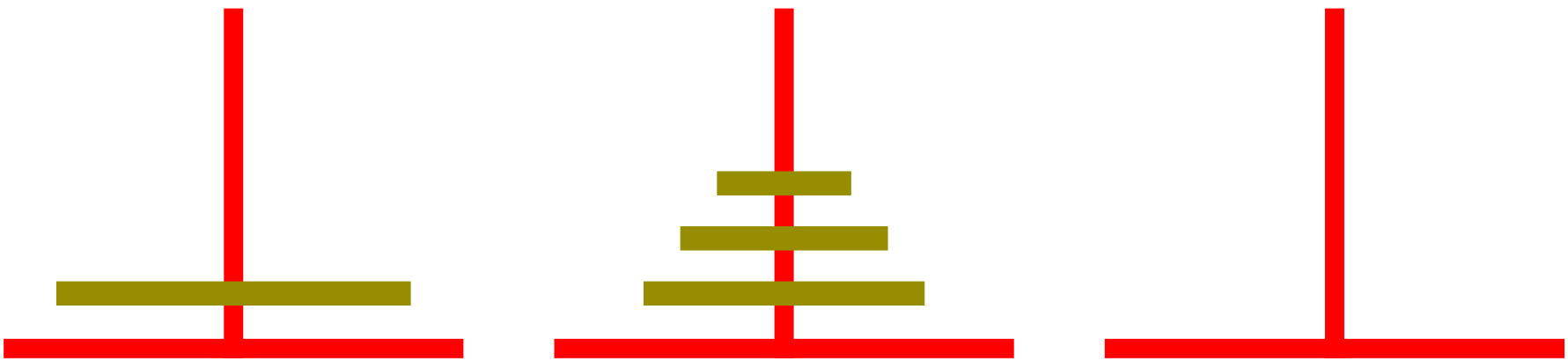
Tour de Hanoï

Etape 6: déplacer un disque de la tige 2 vers la tige 1



Tour de Hanoï

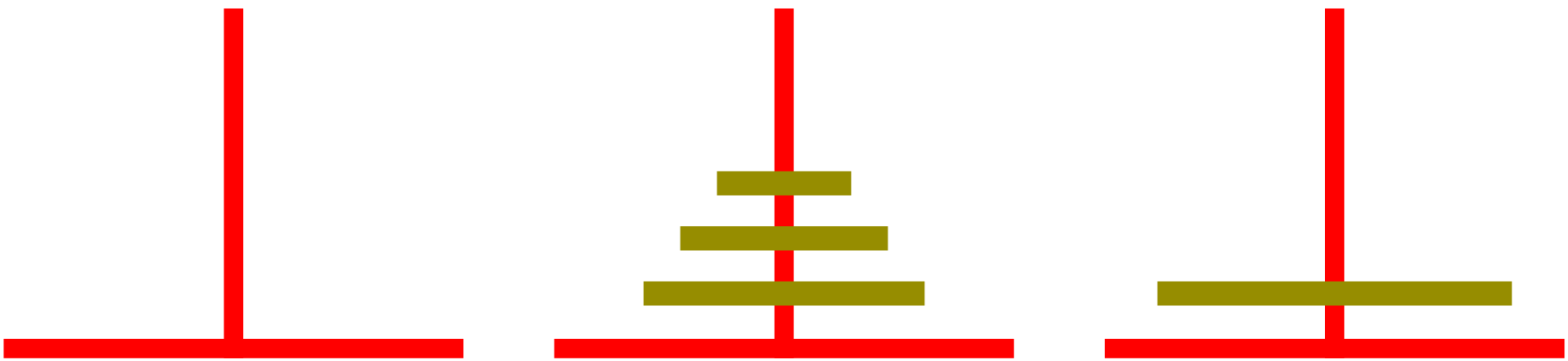
Etape 7: déplacer un disque de la tige 0 vers la tige 1



Tige 0 Tige 1 Tige 2

Tour de Hanoï

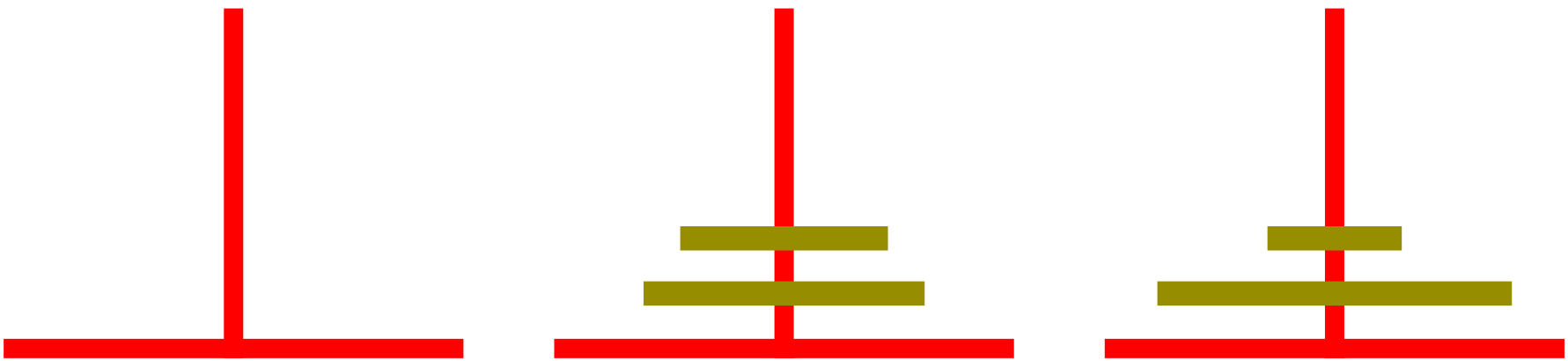
Etape 8: déplacer un disque de la tige 0 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

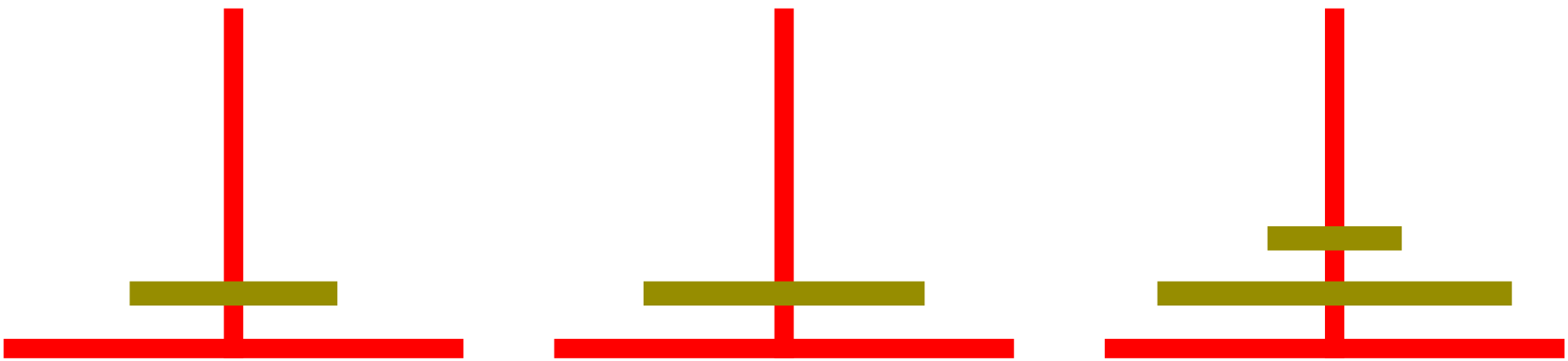
Etape 9: déplacer un disque de la tige 1 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

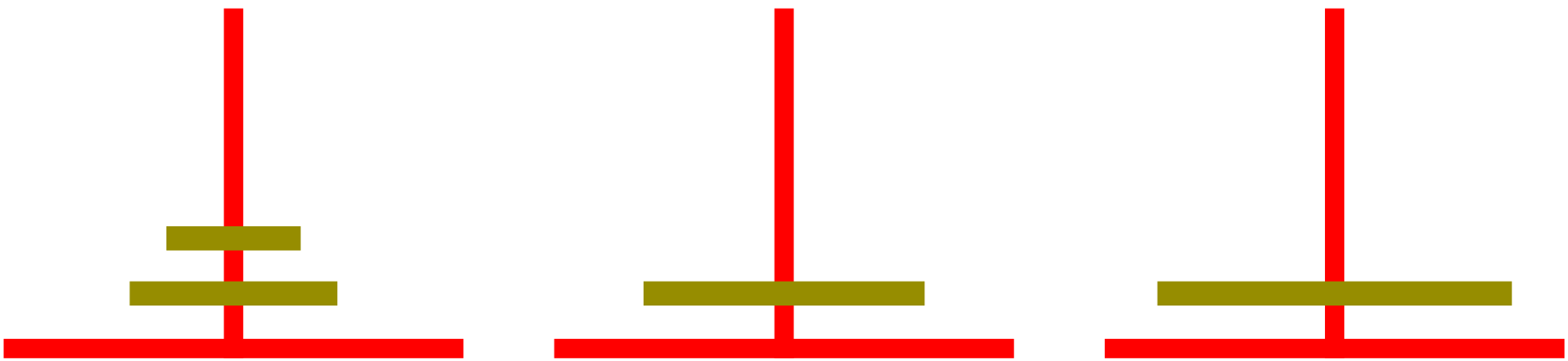
Etape 10: déplacer un disque de la tige 1 vers la tige 0



Tige 0 Tige 1 Tige 2

Tour de Hanoï

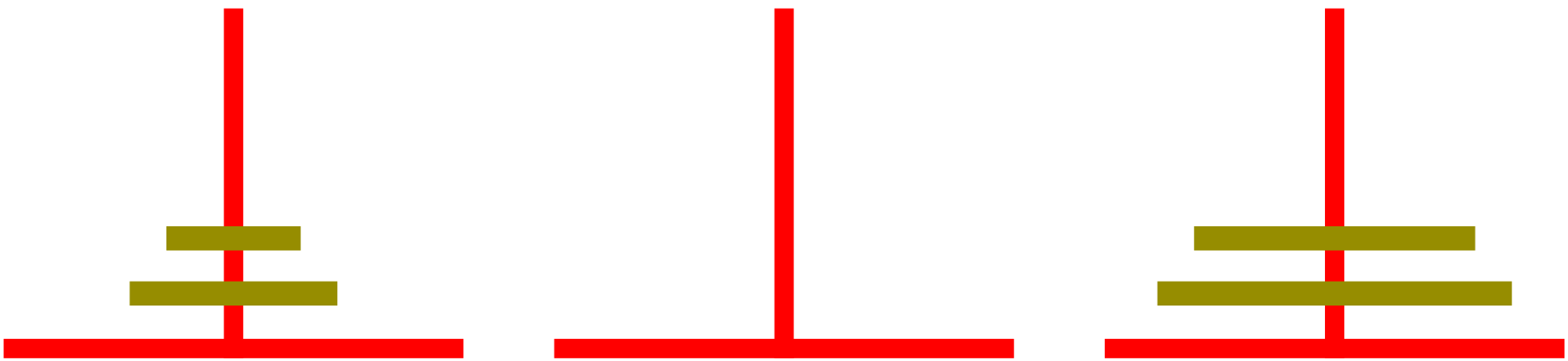
Etape 11: déplacer un disque de la tige 2 vers la tige 0



Tige 0 Tige 1 Tige 2

Tour de Hanoï

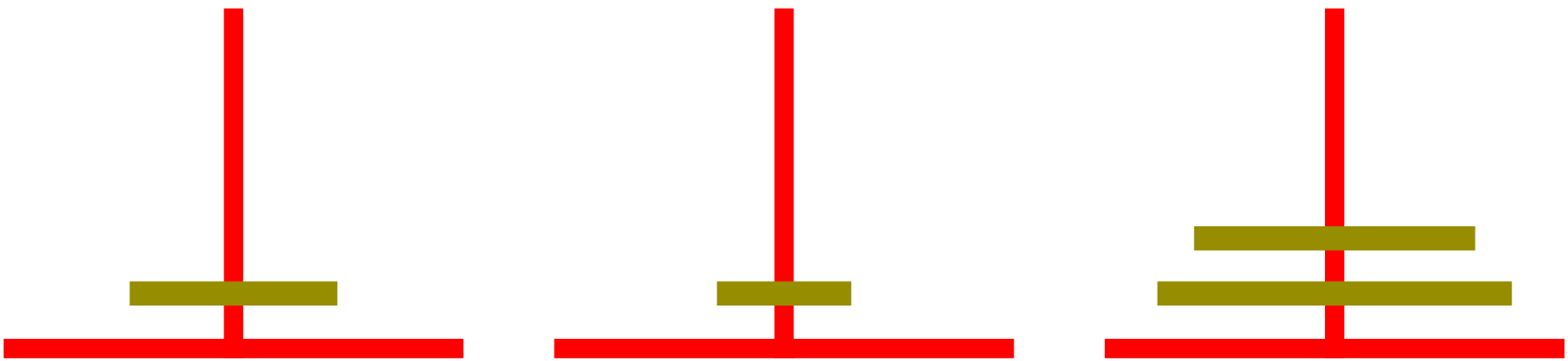
Etape 12: déplacer un disque de la tige 1 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

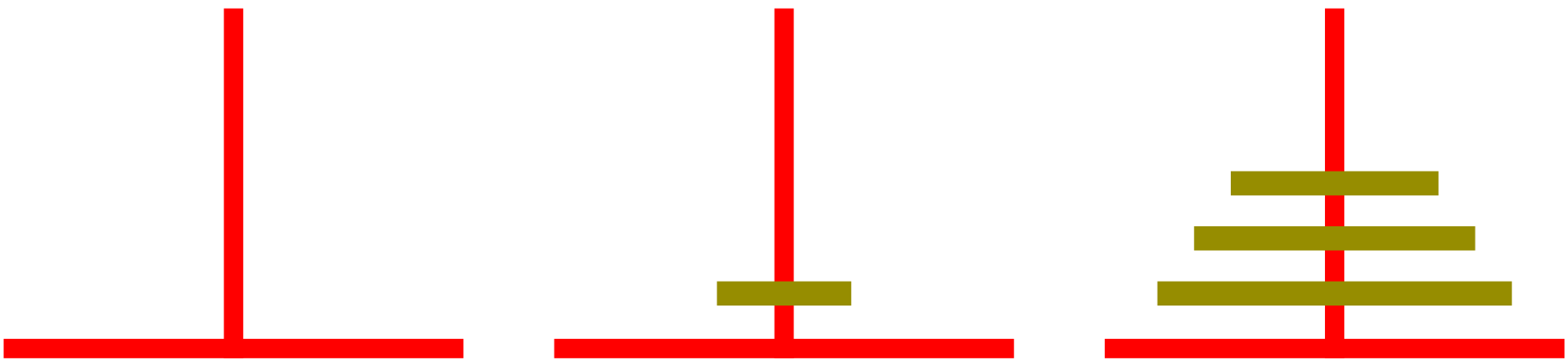
Etape 13: déplacer un disque de la tige 0 vers la tige 1



Tige 0 Tige 1 Tige 2

Tour de Hanoï

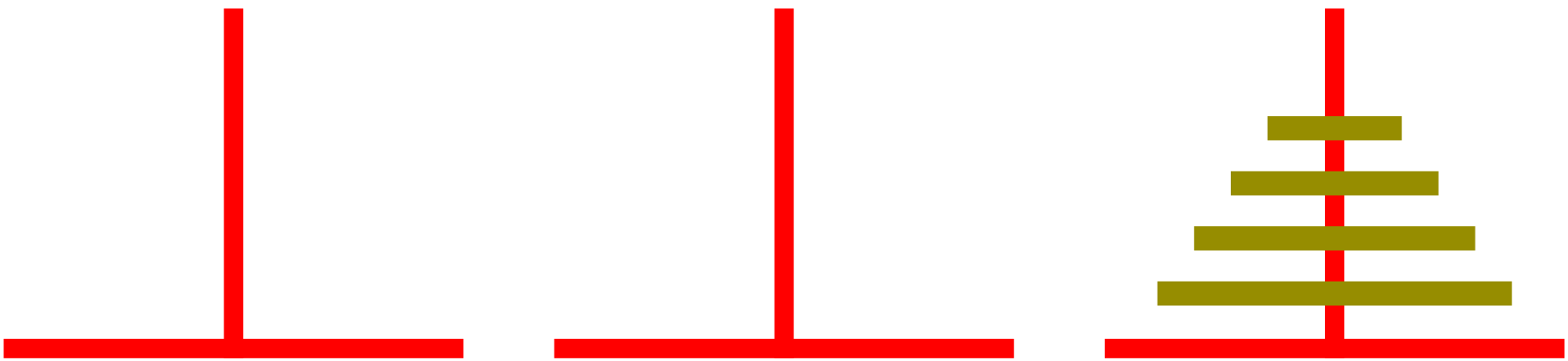
Etape 14: déplacer un disque de la tige 0 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

Etape 15: déplacer un disque de la tige 1 vers la tige 2



Tige 0 Tige 1 Tige 2

Tour de Hanoï

Exercice

Ecrire une fonction récursive qui réalise la fonction de Hanoï

Exécution de la fonction : ce qui est attendu

L'appel suivant (depuis le main):

$$hanoi(3, 0, 1, 2);$$

donnerait la suite d'impressions suivante :

```
Deplacer un disque de 0 vers 2
Deplacer un disque de 0 vers 1
Deplacer un disque de 2 vers 1
Deplacer un disque de 0 vers 2
Deplacer un disque de 1 vers 0
Deplacer un disque de 1 vers 2
Deplacer un disque de 0 vers 2
```

Récursivité : Fonction Hanoi

```
2  void hanoi(int nb_disques, int dep,  int intermediare, int dest)
3  {
4   if(nb_disques==1)
5    printf("Deplacer un disque de %d vers %d\n", dep, dest);
6   else
7   {
8
9
0
1   }
   }
```

Récursivité : Fonction Hanoi

```
2  void hanoi(int nb_disques, int dep,  int intermediare, int dest)
3  {
4   if(nb_disques==1)
5    printf("Deplacer un disque de %d vers %d\n", dep, dest);
6   else
7   {
8    hanoi(nb_disques-1,dep,dest,intermediare);
9
0   }
   }
```

Récursivité : Fonction Hanoi

```
2  void hanoi(int nb_disques, int dep,  int intermediare, int dest)
3  {
4   if(nb_disques==1)
5    printf("Deplacer un disque de %d vers %d\n", dep, dest);
6   else
7   {
8    hanoi(nb_disques-1,dep,dest,intermediare);
9    hanoi(1,dep,intermediare, dest);
0   }
   }
```

Récursivité : Fonction Hanoi

```
2  void hanoi(int nb_disques, int dep,  int intermediare, int dest)
3  {
4   if(nb_disques==1)
5    printf("Deplacer un disque de %d vers %d\n", dep, dest);
6   else
7   {
8    hanoi(nb_disques-1,dep,dest,intermediare);
9    hanoi(1,dep,intermediare, dest);
0    hanoi(nb_disques-1,intermediare,dep,dest);
1   }
2  }
```

Déroulons
la fonction Hanoï

Exécution de la fonction

Voici d'abord le résultat de la fonction avec l'appel suivant (depuis le main):

$$hanoi(3, 0, 1, 2);$$

On obtient la suite d'impressions suivante :

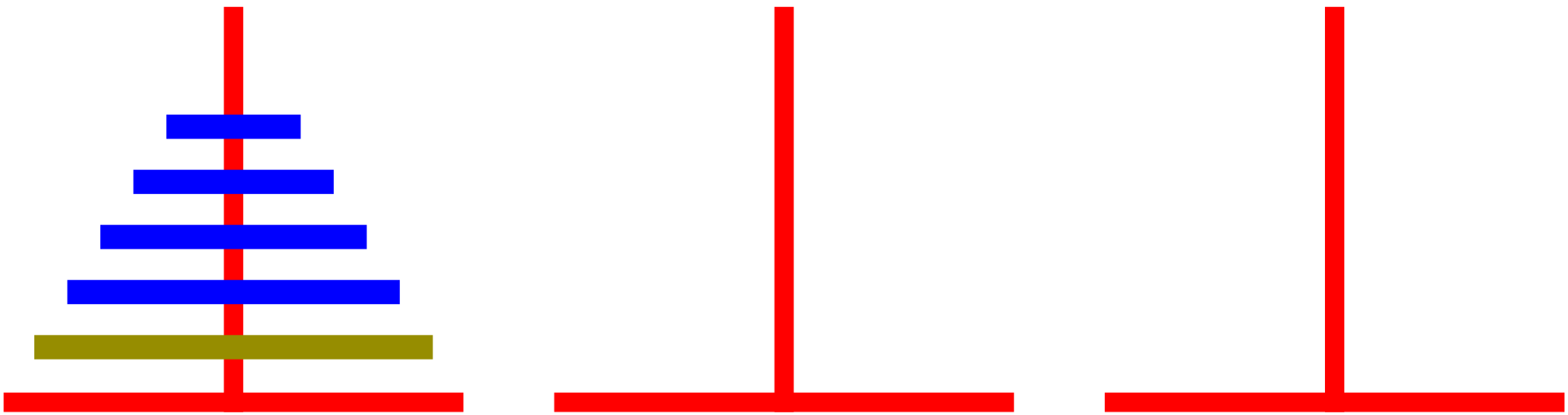
```
Deplacer un disque de 0 vers 2
Deplacer un disque de 0 vers 1
Deplacer un disque de 2 vers 1
Deplacer un disque de 0 vers 2
Deplacer un disque de 1 vers 0
Deplacer un disque de 1 vers 2
Deplacer un disque de 0 vers 2
```

Regardons les différents appels de la fonction Hanoï

Regardons l'impact de chaque instruction la fonction Hanoi

```
2  void hanoi(int nb_disques, int dep,  int intermediaire, int dest)
3  {
4   if(nb_disques==1)
5    printf("Deplacer un disque de %d vers %d\n", dep, dest);
6   else
7   {
8    hanoi(nb_disques-1,dep,dest,intermediaire);
9    hanoi(1,dep,intermediaire, dest);
0    hanoi(nb_disques-1,intermediaire,dep,dest);
    }
2  }
```

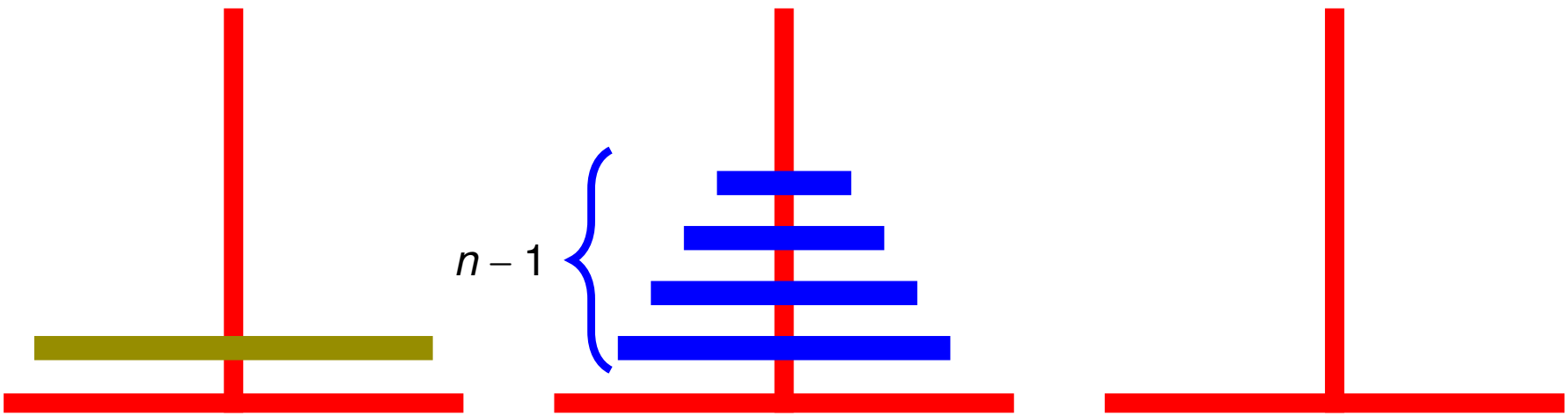
Voici l'état de la tour de Hanoï avant l'exécution de l'algorithme.



Tige 0 (Dep) Tige 1 (intermediaire) Tige 2 (dest)

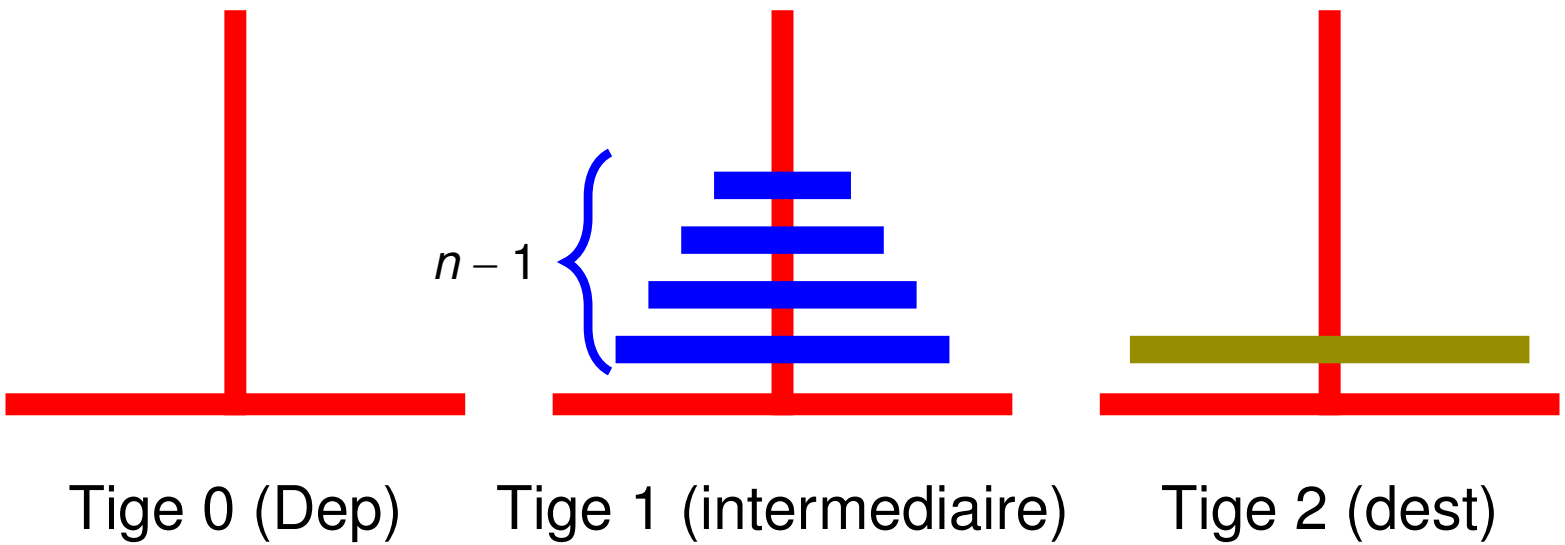
Voici l'état de la tour après l'exécution de l'instruction :

```
2  void hanoi(int nb_disques, int dep,  int intermediaire, int dest)
3  {
4    ...
5     hanoi(nb_disques-1,dep,dest,intermediaire);
6    ...
7  }
```



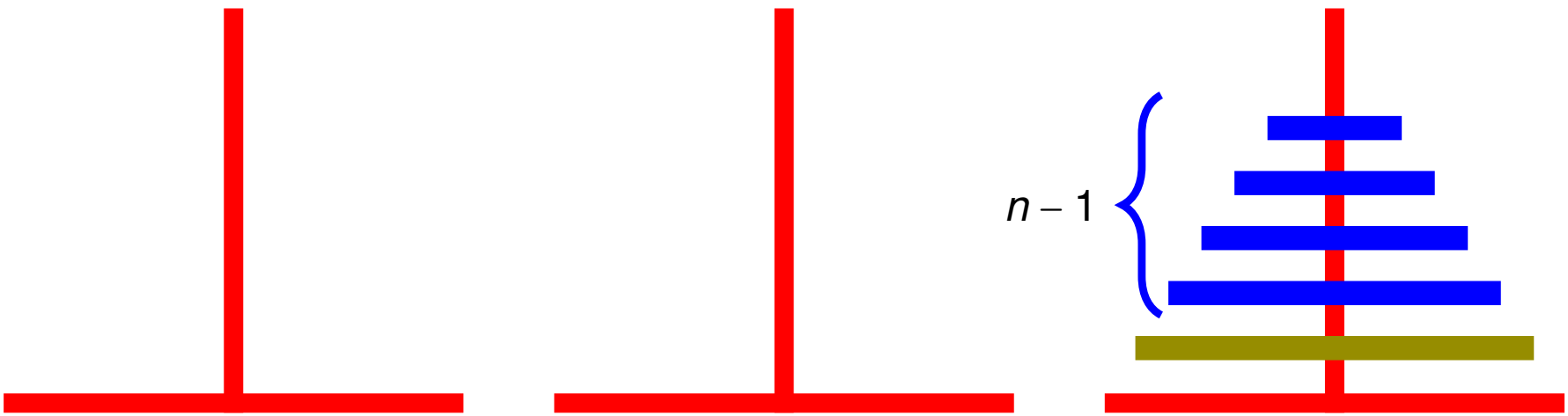
Voici l'état de la tour après l'exécution de l'instruction :

```
2  void hanoi(int nb_disques, int dep,  int intermediaire, int dest)
3  {
4   if(nb_disques==1)
5    printf("Deplacer un disque de %d vers %d\n", dep, dest);
6   else
7   {
8    ...
9    hanoi(1,dep,intermediaire, dest);
0    ...
     }
2  }
```



Voici l'état de la tour après l'exécution de l'instruction :

```
2  void hanoi(int nb_disques, int dep,  int intermediaire, int dest)
3  {
4  ...
5   hanoi(nb_disques-1,intermediaire,dep,dest);
6  ...
7  }
```



Tige 0 (Dep) Tige 1 (intermediaire) Tige 2 (dest)

Complexité temporelle

de la fonction Hanoï

Première approche

- Essayer de deviner une forme générale de la complexité temporelle d'une fonction
  - Par exemple, on peut prendre des variables qui calculent le nombre d'appels récursifs.
- Le vérifier pour des arguments quelconques (par récurrence)

Complexité : Tour de Hanoï

```
void hanoi(int nb_disques, dep,  intermediaire, dest)
{


 nb_appels_recursif ++;

 if(nb_disques==1)
  printf("Deplacer un disque de %d vers %d\n", dep, dest);
 else
 {
  hanoi(nb_disques-1,dep,dest,intermediaire);
  printf("Deplacer un disque de %d vers %d\n",dep,dest);
  hanoi(nb_disques-1,intermediaire,dep,dest);
 }
}
```

Complexité : Tour de Hanoï

nombre de disques (n)	Nombre d'appels récursifs T(n)
1	1
2	3
3	7
4	15
5	31
6	63
7	127
8	255

Exercice

Quelle est la relation entre T(n) et n?

Complexité : Tour de Hanoï

nombre de disques (n)	Nombre d'appels récursifs T(n)
1	1
2	3
3	7
4	15
5	31
6	63
7	127
8	255

Exercice

On remarque que :

$$T(n) = 2^n - 1.$$

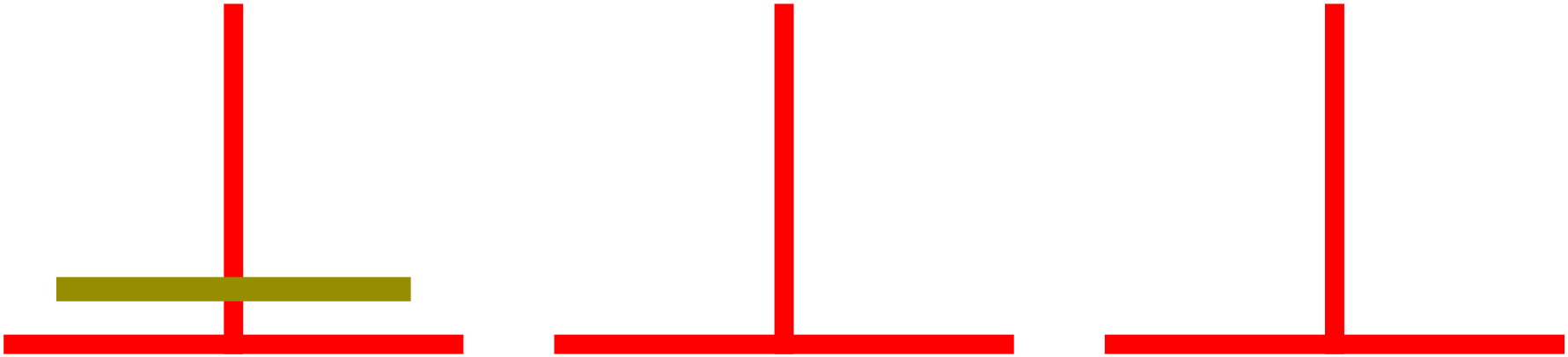
Il faut confirmer cette relation par récurrence.

Complexité : Tour de Hanoï

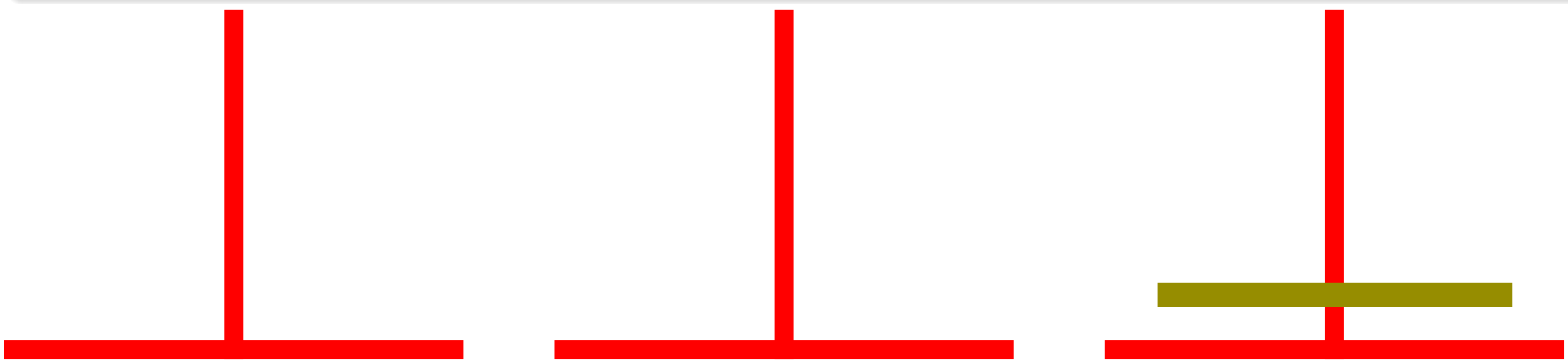
Remarques

- Calculer le nombre d'appels récursifs (ou d'instructions élémentaires) permet, dans certains cas, d'obtenir une idée sur la complexité temporelle d'une fonction.
- Ca reste clairement insuffisant.
- Il est important de regarder de près l'algorithme et déterminer l'équation qui relie T(n) en fonction de T(n-1), T(n-2) etc

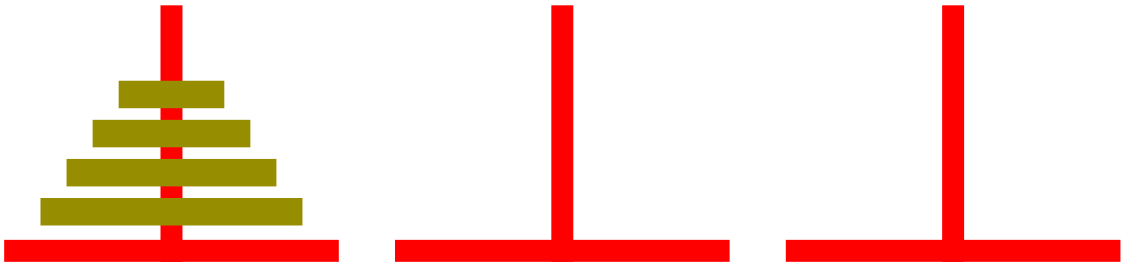
Regardons de près l'algorithme de Hanoï N=1



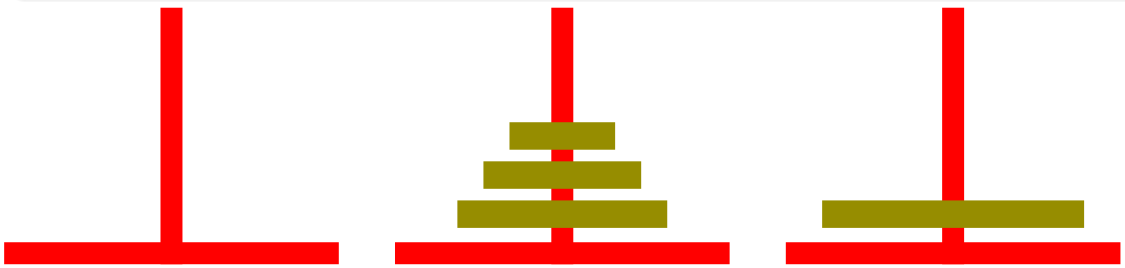
Déplacer le disque de 1 vers 3



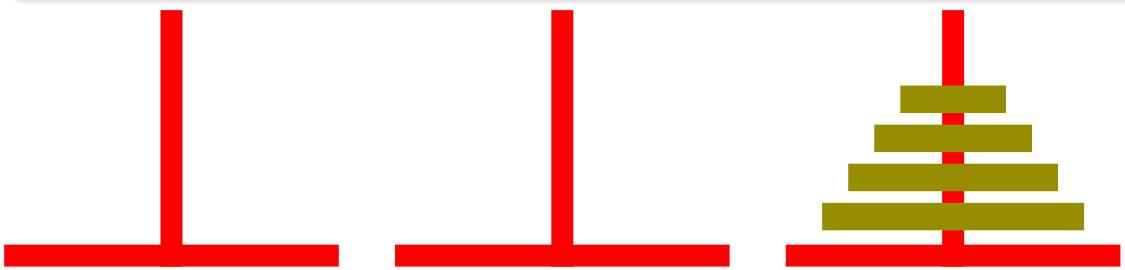
Regardons de près l'algorithme de Hanoï (cas général)



Déplacer $n-1$ disques de 1 vers 2 , puis 1 disque de 1 vers 3



Déplacer $n-1$ disques de 2 vers 3



Regardons de près l'algorithme de Hanoï

Pour résumer

- $T(1) = 1$
- Pour N différent de 1, il y a trois étapes consécutives :
  - Déplacer $N-1$ premiers disques de 1 vers 2, avec un complexité de $T(n-1)$
  - Déplacer un disque de 1 vers 3, avec un complexité de 1
  - Déplacer les $N-1$ disques de 2 vers 3, avec un complexité de $T(n-1)$
- Ce qui donne au final :

$$T(n) = 2.T(n-1) + 1.$$

Regardons de près l'algorithme de Hanoï

Méthode par substitution

L'idée est de développer progressivement $T(n)$ jusqu'à atteindre des cas de base $T(0) = 0$ ou $T(1) = 1$.
Dans l'exemple de la tour de Hanoï (standard), nous avons :

$$\begin{aligned}
T(n) &= 2.T(n-1)+1 \\
&= 2.(2.T(n-2)+1)+1 \\
&= 2^2.T(n-2)+(1+2) \\
&= 2^3.T(n-3)+(1+2+2^2) \\
&\quad . \\
&\quad . \\
&= 2^{n-1}.T(1)+(1+2+2^2+\ldots+2^{n-2}) \\
&= 1+2+2^2+\ldots+2^{n-2}+2^{n-1} \\
&= 2^n-1.
\end{aligned}$$

Récursivité Croisée

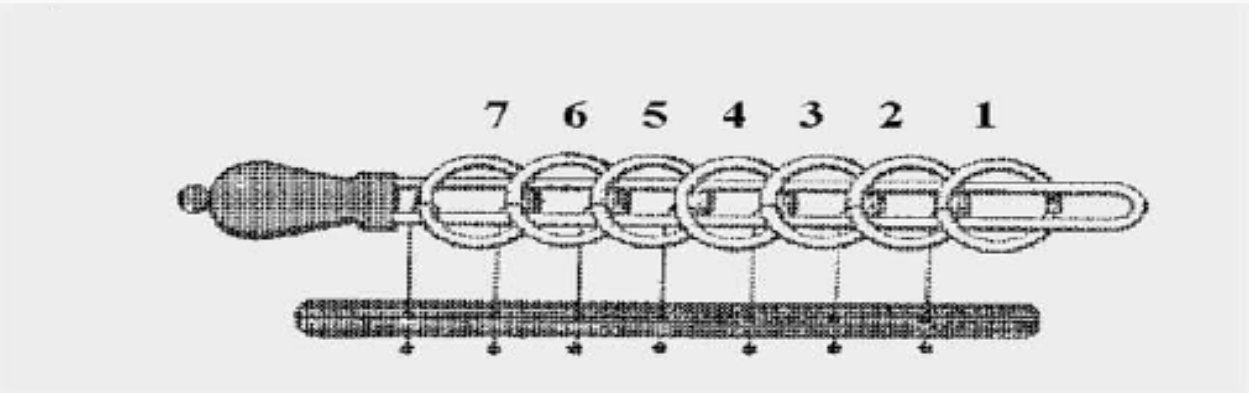
Type de récursivité

Récursivité croisée

Deux fonctions sont dites mutuellement récursives si elles dépendent les unes des autres.

Type de récursivité

Baguenaudière

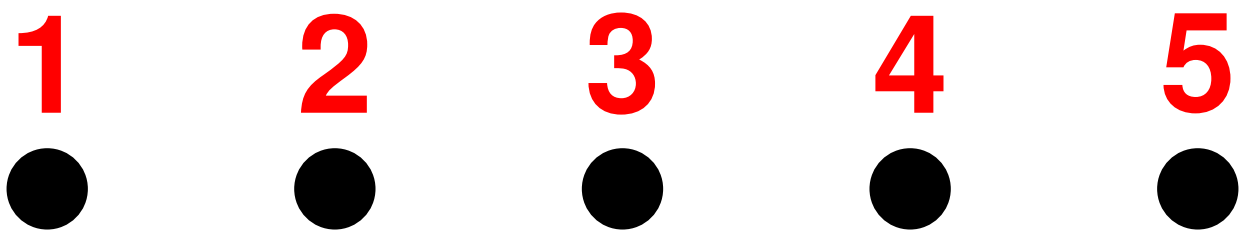


Récursivité : Baguenaudière

Baguenaudière : principe

On se donne une table (que l'on appellera baguenaudière) à n cases, chacune peut contenir un pion. Chaque case est numérotée de "1" à "n". La baguenaudière peut être vide ou pleine.

Exemple : Une baguenaudière pleine à 5 cases



Récursivité : Baguenaudière

Baguenaudière : Règles du jeu

- Pour la case 1, on peut enlever un pion ou mettre un pion sans contrainte.
- Pour la case 2, on peut enlever un pion ou mettre un pion uniquement si la case 1 est pleine
- De manière générale, pour la case i (différente de 1 et 2), on peut enlever un pion ou mettre un pion si :
  - La case $i - 1$ est pleine (contient un pion)
  - Toutes les cases de 1 à $i - 2$ sont vides.

Récursivité : Baguenaudière

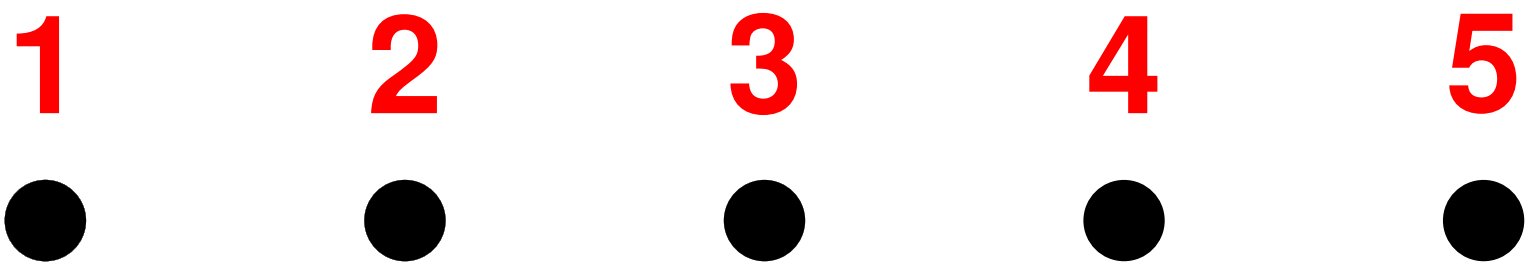
Baguenaudière : But du jeu

- La baguenaudière peut être vide ou pleine.
- Si elle est vide, le but est de la remplir
- Si elle est pleine, le but est de la vider

Exemple de baguenaudière

Remplir 5 cases

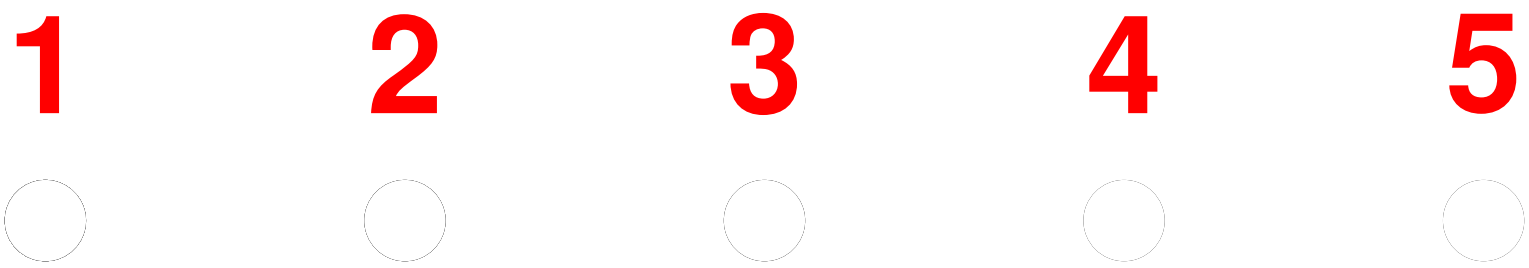
Déroulement du Jeu Baguenaudiere : remplir 5 cases



Exemple de baguenaudière

Vider 5 cases

Déroulement du Jeu Baguenaudiere : vider 5 cases



Récursivité : Baguenaudière

Exercices

- On rappelle qu'une baguenaudière peut être vide ou pleine.
- Ecrire une fonction récursive qui consiste à remplir une baguenaudière vide.
- Ecrire une fonction récursive qui consiste à vider une baguenaudière pleine.

Baguenaudière : ce qui est attendu du programme :

Si on appelle la fonction Vider(4) depuis le main on obtient la suite d'impressions suivante :

```
Enlever un pion de la case 2
Enlever un pion de la case 1
Enlever un pion de la case 4
Mettre un pion dans la case 1
Mettre un pion dans la case 2
Enlever un pion de la case 1
Enlever un pion de la case 3
Mettre un pion dans la case 1
Enlever un pion de la case 2
Enlever un pion de la case 1
```

Baguenaudière : ce qui est attendu du programme :

Si on appelle la fonction Remplir(4) depuis le main on obtient la suite d'impressions suivante :

```
Mettre un pion dans la case 1
Mettre un pion dans la case 2
Enlever un pion de la case 1
Mettre un pion dans la case 3
Mettre un pion dans la case 1
Enlever un pion de la case 2
Enlever un pion de la case 1
Mettre un pion dans la case 4
Mettre un pion dans la case 1
Mettre un pion dans la case 2
```

Solutions

Récursivité : Fonction remplir

```
2  void Remplir (int N)
3  {
4    if (N>0)
5    {
6      if (N==1) {
7      printf("Mettre un pion dans la case %d \n", N);
8      }
9  ....
0    }
   }
```

Récursivité : Fonction remplir

Le but est de placer le n^{me} pion.

```
void Remplir (int N)
{
  if (N>0)
  {
    if (N==1) {
    printf("Mettre un pion dans la case %d \n", N);
    }
    else
    {
    Remplir(N-1);
....
    }
  }
}
```

Récursivité : Fonction remplir

```
void Remplir (int N)
{
  if (N>0)
  {
    if (N==1) {
    printf("Mettre un pion dans la case %d \n", N);
    }
    else
    {
    Remplir(N-1);
    Vider (N-2);
....
    }
  }
}
```

Récursivité : Fonction remplir

```
void Remplir (int N)
{
  if (N>0)
  {
    if (N==1) {
    printf("Mettre un pion dans la case %d \n", N);
    }
    else
    {
    Remplir(N-1);
    Vider (N-2);
    printf("Mettre un pion dans la case %d \n", N);
...
    }
  }
}
```

Récursivité : Fonction remplir

```
void Remplir (int N)
{
  if (N>0)
  {
    if (N==1) {
    printf("Mettre un pion dans la case %d \n", N);
    }
    else
    {
    Remplir(N-1);
    Vider (N-2);
    printf("Mettre un pion dans la case %d \n", N);
    Remplir(N-2);
    }
  }
}
```

Récursivité : Fonction vider

```
2  void Vider (int N)
3  {
4    if (N>0)
5    {
6      if (N==1) {
7        printf( "Enlever un pion de la case %d \n", N);
8      }
9  ...
0    }

2  }
```

Récursivité : Fonction vider

```
void Vider (int N)
{
  if (N>0)
  {
    if (N==1) {
      printf( "Enlever un pion de la case %d \n", N);
    }
    else
    {
      Vider (N-2);
....
    }
  }

}
```

Récursivité : Fonction vider

```
void Vider (int N)
{
  if (N>0)
  {
    if (N==1) {
      printf( "Enlever un pion de la case %d \n", N);
    }
    else
    {
      Vider (N-2);
      printf("Enlever un pion de la case %d \n", N);
....
    }
  }

}
```

Récursivité : Fonction vider

```
void Vider (int N)
{
  if (N>0)
  {
    if (N==1) {
      printf( "Enlever un pion de la case %d \n", N);
    }
    else
    {
      Vider (N-2);
      printf("Enlever un pion de la case %d \n", N);
      Remplir(N-2);
....
    }
  }
}
```

Récursivité : Fonction vider

```
void Vider (int N)
{
  if (N>0)
  {
    if (N==1) {
      printf( "Enlever un pion de la case %d \n", N);
    }
    else
    {
      Vider (N-2);
      printf("Enlever un pion de la case %d \n", N);
      Remplir(N-2);
      Vider(N-1);
    }
  }

}
```