Partie 2

Récursivité

2 octobre 2018

Plan

1. Principe

2. Exemples
   - Tours de Hanoï
   - Nombres de Fibonacci
   - Fonction puissance

3. Implémentation de la récursivité

Plan

Généralités

- En programmation, la majorité des problèmes (non trivaux) nécessitent de répéter une séquence d'instructions selon un schéma dépendant des données.
- Deux types de solutions algorithmiques :
  - Algorithmes **itératifs**, basés sur des boucles (Partie 1).
  - Algorithmes **récursifs**, basés sur des fonctions qui s'invoquent elles-mêmes.
- Les deux types de solutions sont toujours possibles mais une solution récursive est parfois plus simple et naturelle qu'une solution itérative (et inversement).

Définition

- Un algorithme de résolution d'un problème P sur une donnée a est dit récursif si parmi les opérations utilisées pour le résoudre, on trouve une résolution du même problème P sur une donnée b.
- Dans un algorithme récursif, on nommera appel récursif toute étape résolvant le même problème sur une autre donnée.
- Un algorithme récursif s'implémente généralement via des fonctions dont l'exécution conduit à leur propre invocation. On appellera ces fonctions des fonctions récursives.
- Forme générale d'une fonction récursive (directe) :

```
type f(P) {
  ...
  x = f(Pr);
  ...
  return r;
}
```

Illustration 1 : fonction factorielle

La définition mathématique de la factorielle est récursive :

$$n! = \begin{cases} 1 & \text{si } n \leq 1, \\ n * (n-1)! & \text{sinon} \end{cases}$$

et se prête donc naturellement à une implémentation via une fonction récursive :

```
int fact(int n) {
  if (n <= 1)
     return 1;

  return n * fact(n-1);
}
```

Illustration 1 : fonction factorielle

Trace des appels de fonctions pour `fact(5)` :

```
fact(5)
  |fact(4)
  |  |fact(3)
  |  |  |fact(2)
  |  |  |  |fact(1)
  |  |  |  |  |return 1
  |  |  |  |return 2*1 = 2
  |  |  |return 3*2 = 6
  |  |return 4*6 = 24
  |return 5*24 = 120
```

Illustration 2 : algorithme d'Euclide

L'algorithme du calcul du pgcd d'Euclide est basé sur la propriété récursive suivante :

> *Soient deux entiers positifs a et b. Si $a > b$, le pgcd de a et b est égal au pgcd de b et de $(a \mod b)$.*

Suggère l'implémentation récursive suivante :

```
int pgcd(int a, int b) {
  if (b > a)
    return pgcd(b,a);

  if (b == 0)
    return a;

  return pgcd(b, a % b);
}
```

Exemple :

```
pgcd(1440, 408)
  |pgcd(408, 216)
  |  |pgcd(216, 192)
  |  |  |pgcd(192, 24)
  |  |  |  |pgcd(24,0)
  |  |  |  |  |return 24
  |  |  |  |return 24
  |  |  |return 24
  |  |return 24
  |return 24
```

Conception de fonctions récursives

Deux conditions pour qu'une fonction récursive soit bien définie :

- Présence d'un cas de base(condition d'arrêt)
- La "taille" du problème doit être réduite à chaque étape

Forme générale d'une fonction récursive :

```
type f(P) {
  if (cas_de_base) { // cas de base
    ... // pas d'appel récursif
    return [...];
  }

  // étape de réduction
  ...
  [x =] f(Pr); // avec Pr plus "simple" que P
  ...
  [return r;]
}
```

Conception de fonctions récursives

D'autres formes peuvent néanmoins être valides (mais sont plutôt à éviter).

Récursivité non décroissante

(Suite de Syracuse)

```
int Syracuse(int n) {
  if (n == 1)
     return 1;

  if (n % 2) // n est impair
     return Syracuse(3*n+1);
  else       // n est pair
     return Syracuse(N/2);
}
```

Récursivité imbriquée

(fonction d'Ackermann)

```
int ack(int m, int n) {
  if (m==0)
     return n+1;
  else {
     if (n==0)
        return ack(m-1,1);
     else
        return ack(m-1,ack(m,n-1));
  }
}
```

Récursivité croisée

```
int P(int n) {
  if (n==0)
     return 1;
  else
     return I(n-1);
}

int I(int n) {
  if (n==0)
     return 0;
  else
     return P(n-1);
}
```

Que calcule la fonction P ?

Correction formelle d'algorithmes récursifs

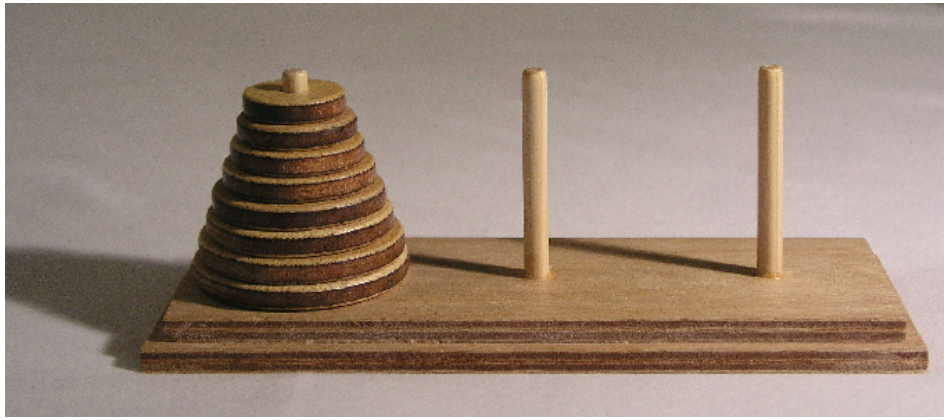
```
int fact(int n) {
  if (n <= 1)
     return 1;

  return n * fact(n-1);
}
```

La correction d'un algorithme récursif se prouve pas induction :

- On montre que le code est correct dans le cas de base.
  - Si $n \leq 1$, `fact`(n) renvoie 1, ce qui est correct.
- On montre que l'étape de réduction est correcte en supposant que les appels récursifs sont corrects (hypothèse inductive)
  - Si $n > 1$, la fonction renvoie $n *$ `fact`$(\mathtt{n} - 1)$, qui vaut bien $n!$ si `fact(n-1)` renvoie $(n-1)!$.
- On en conclut, par le *principe d'induction*, que l'algorithme est correct pour toutes les entrées.

Plan

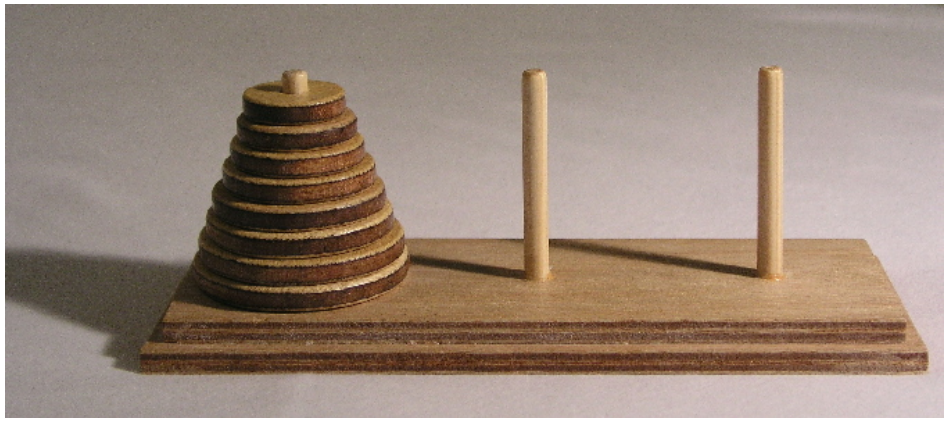
Un classique : le problème des tours de Hanoï



Source : wikipedia

- On considère un jeu comprenant trois tiges sur lesquelles sont empilés un certain nombre n de disques de diamètres différents.
- Initialement, tous les disques sont empilés sur la première tige, par ordre décroissant de diamètre.
- L'objectif est d'amener tous les disques sur une autre tige sous les deux contraintes suivantes :
  1. On ne peut déplacer qu'un seul disque à la fois.
  2. On ne peut poser un disque que sur un disque de diamètre supérieur (ou sur le support).

Un classique : le problème des tours de Hanoï

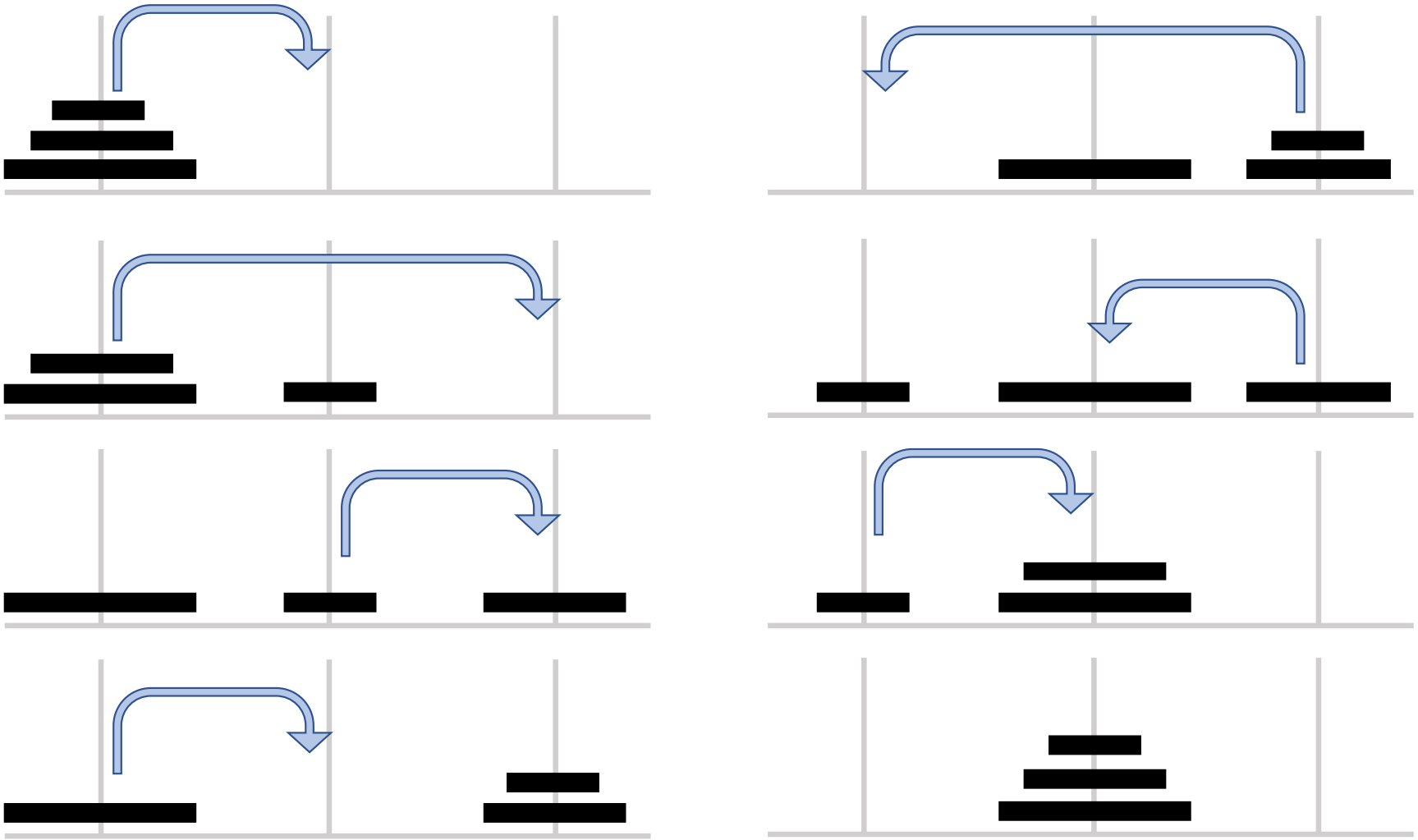


Source : wikipedia

- Objectif : écrire un programme générant la séquence de mouvements permettant d'atteindre l'objectif, pour un nombre n donné de disques.
- Seulement deux types de mouvements possibles pour un disque donné :
  - à Droite : de 1 à 2, de 2 à 3 ou de 3 à 1.
  - à Gauche : de 1 à 3, de 2 à 1, ou de 3 à 2.
- Séquence de mouvements affichée sous la forme : "1G 2D 1G..." (numéro de disque + type de mouvement).

Tours de Hanoï : solution pour $n = 3$

1D 2G 1D 3D 1D 2G 1D



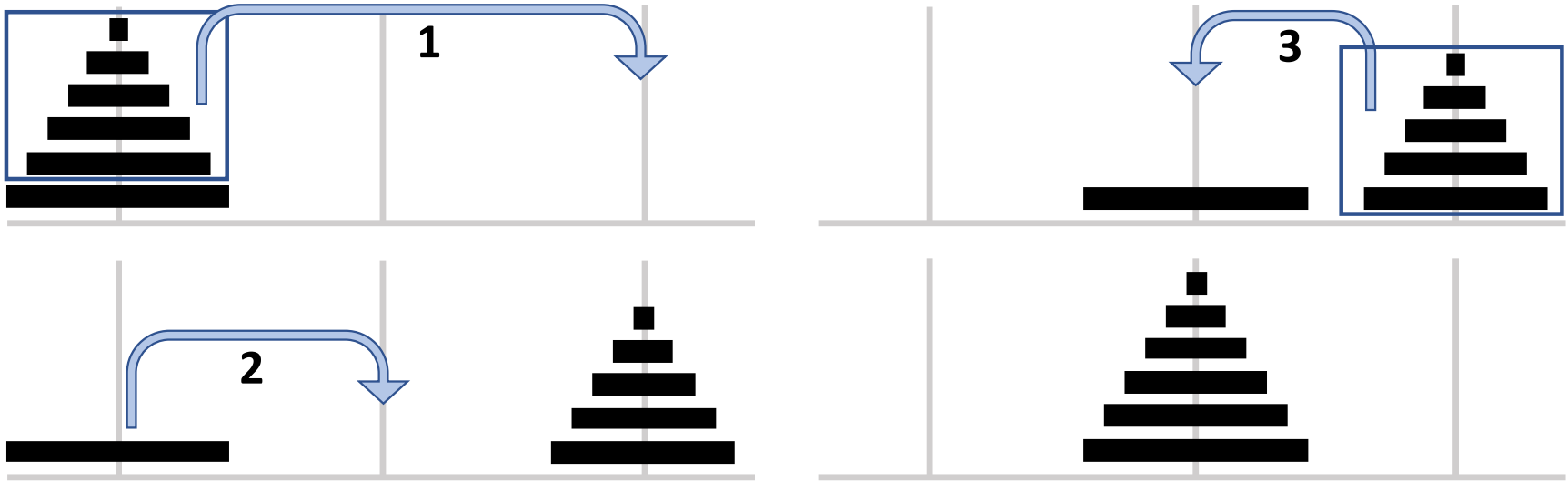
Principe de construction d'un algorithme récursif

- Trouver un ou plusieurs paramètres de taille sur lesquels construire la récursion

  Hanoï : n, le nombre de disques
- Trouver une solution pour le(s) cas de base, c'est-à-dire des problèmes de petites tailles (la plupart du temps trivial).

  Hanoï : Soit
  - $n = 1 \Rightarrow$ on déplace le disque vers la bonne tige.
  - $n = 0 \Rightarrow$ on ne fait rien.
- Trouver comment réduire le problème à un ou plusieurs sous-problème de tailles strictement plus petites.
  Étape la plus délicate. Revient à trouver l'invariant dans le cas d'algorithmes itératifs.

Tours de Hanoï : une solution récursive



En supposant qu'on puisse déplacer (récursivement) une pile de $n-1$ disques, on peut déplacer une pile de n disques à droite en trois étapes :

1. On déplace les $n-1$ disques vers la gauche
2. On déplacer le $n$ème disque (le plus grand) vers la droite
3. On déplace les $n-1$ disques vers la gauche

Tours de Hanoï : code C

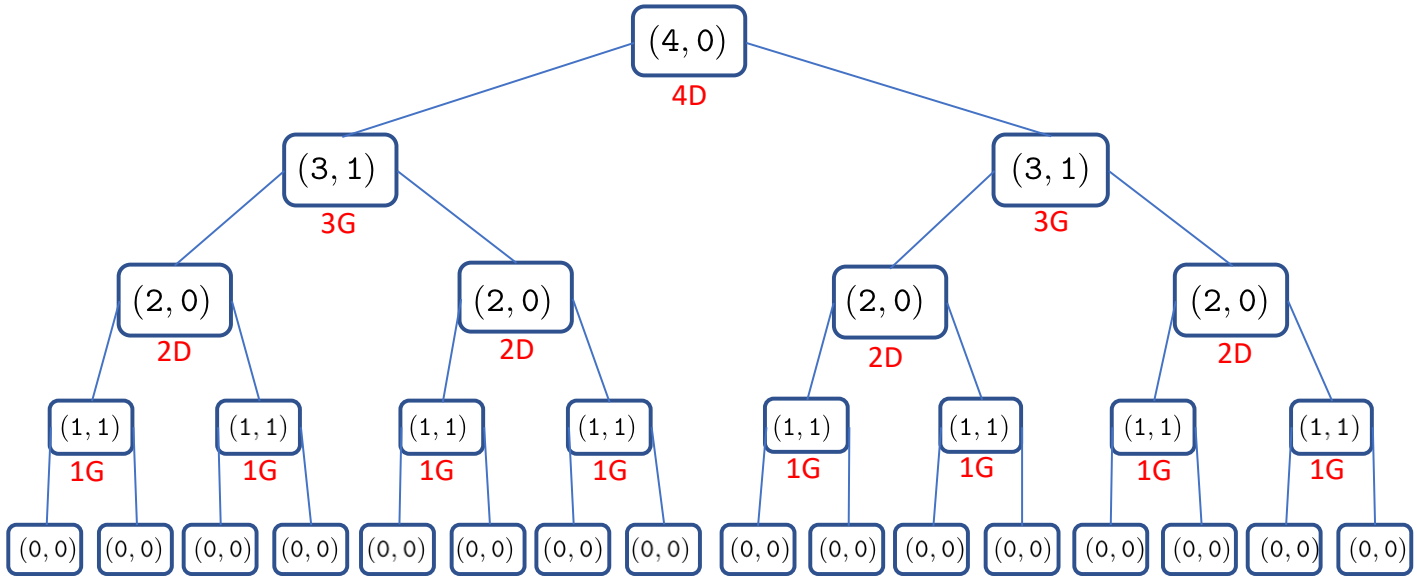
```c
#include <stdio.h>
#include <stdlib.h>

void hanoi(int n, int left) {

  if (n==0)
    return;

  hanoi(n-1,!left);

  if (left)
    printf("%dG\n", n);
  else
    printf("%dD\n", n);

  hanoi(n-1, !left);

}
```

```c
int main(int argc, char *argv[]) {

  int n = atoi(argv[1]);

  // Pour déplacer la pile à droite
  hanoi(n,0);

  return 0;

}
```

Tours de Hanoï : trace des appels récursifs

```
hanoi(4,0)
|hanoi(3,1)
|   |hanoi(2,0)
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|   |   |printf("2D")
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|   |printf("3G")
|   |hanoi(2,0)
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|   |   |printf("2D")
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|printf("4D")
|hanoi(3,1)
|   |hanoi(2,0)
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|   |   |printf("2D")
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|   |printf("3G")
|   |hanoi(2,0)
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
|   |   |printf("2D")
|   |   |hanoi(1,1)
|   |   |   |printf("1G")
```

Dans le cas d'une récursivité multiple, on peut visualiser les appels récursifs selon un arbre.



Tours de Hanoï : algorithme itératif

A partir de l'arbre, on peut observer les propriétés suivantes (qui peuvent se démontrer par induction) :

- Le premier mouvement et puis un mouvement sur deux implique le plus petit disque qui va toujours dans la même direction
- Après un mouvement du plus petit disque, il n'y a toujours qu'un seul mouvement possible n'impliquant pas le plus petit disque.

Suggère l'algorithme itératif suivant :

Tant que la pile n'est pas dans sa position finale :

- *Bouger le plus petit disque à gauche/droite*
- *Bouger le seul autre disque qui peut être bougé*

Cet algorithme est strictement identique au récursif mais plus difficile à imaginer à partir de rien.

Tours de Hanoï : complexité

De l'arbre, on peut déduire également que pour n disques, il faudra $2^n - 1$ mouvements.

- $1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$

On peut montrer qu'il n'y a pas moyen de faire mieux.

En conséquence, il n'est possible de résoudre le jeu, que ce soit physiquement ou sur ordinateur, que pour des valeurs de n faibles.

Par exemple pour $n = 64$:

- 1s par mouvement $\Rightarrow$ 584 milliards d'années
- 10^{-9}s par mouvement $\Rightarrow$ 584 années.

Plan

Nombres de Fibonacci

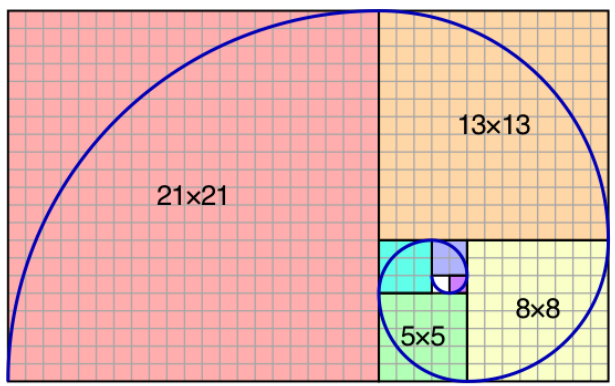
Les nombres de Fibonacci sont définis par la récurrence suivante :

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ \forall n \geq 2 : F_n &= F_{n-2} + F_{n-1} \end{aligned}$$

Apparaissent dans l'analyse de plusieurs phénomènes naturels et dans le domaine de l'art et de l'architecture.

Quand n grand, $F_n \approx \frac{\phi^n}{\sqrt{5}}$, avec $\phi = \frac{1+\sqrt{5}}{2}$ le nombre d'or.

Spirale d'or



Source : wikipedia

Nombres de Fibonacci : implémentation récursive

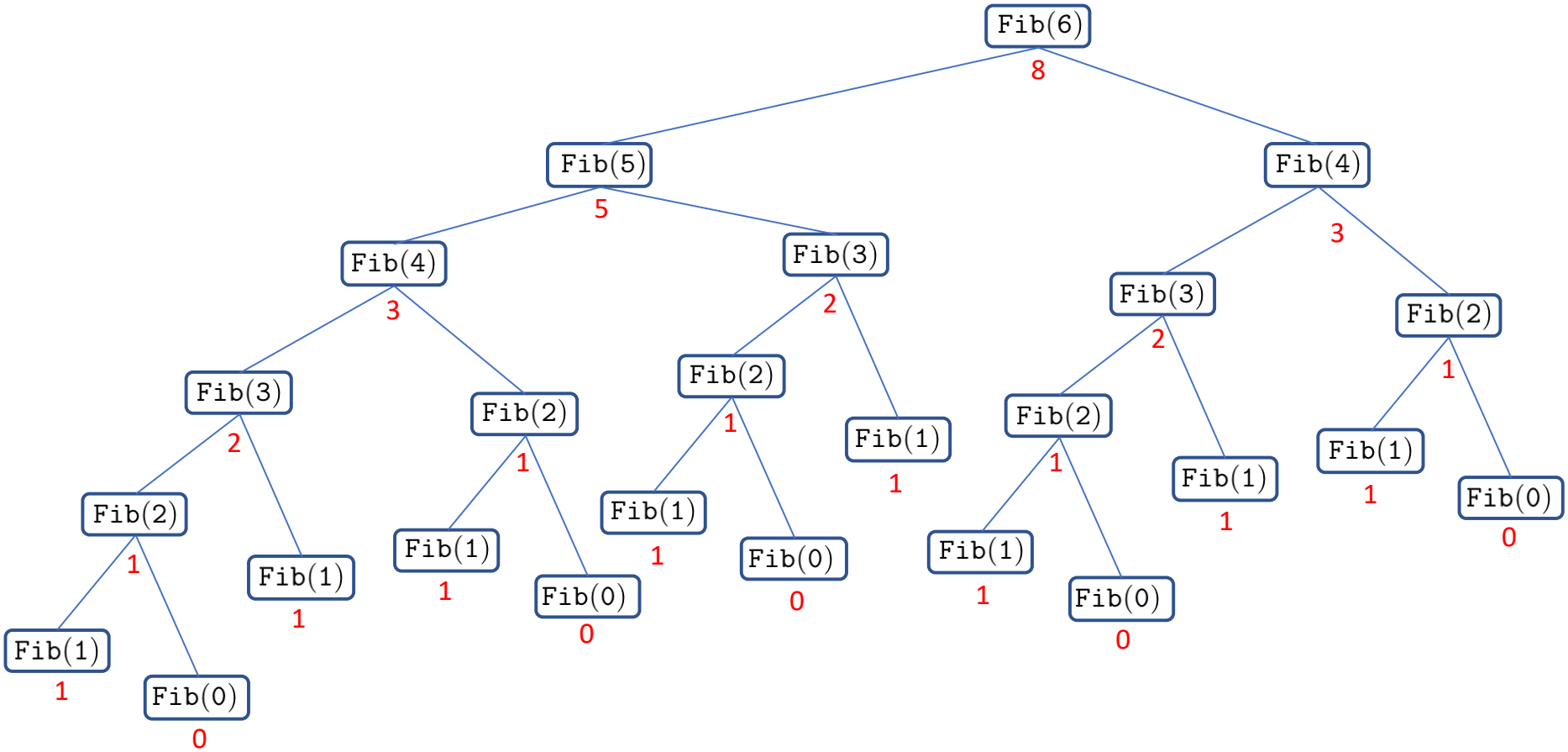
Implémentation récursive directe :

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

Peut-on calculer `Fib(60)` ?

n	temps de calcul
10	0s
20	0s
40	1s
45	8s
50	87s
60	??

Nombres de Fibonacci : arbres des appels récursifs



Beaucoup de valeurs sont recalculées inutilement.

Nombres de Fibonacci : temps de calcul

Soit $T(n)$ le nombre de noeuds de l'arbre des appels récursifs. On a :

$$\begin{aligned} T(0) &= 1 \\ T(1) &= 1 \\ \forall n \geq 2 : T(n) &= T(n-1) + T(n-2) + 1 \end{aligned}$$

Le nombre d'appels de fonctions est donc plus élevé que le $n$ème nombre de Fibonacci qui vaut approximativement $\frac{\phi^n}{\sqrt{5}}$, avec $\phi \approx 1,618$.

Chaque appel de fonction demandant un temps constant, les temps de calcul vont augmenter exponentiellement avec n. Ils sont multipliés par $1,618$ au moins à chaque incrément de n.

S'il faut 87s pour $n = 50$, il faudra au moins 3 heures pour $n = 60$ et 77000 années pour $n = 100$.

Nombres de Fibonacci : version itérative

Une version itérative beaucoup plus efficace

```
int Fibiter(int n) {
  if (n <= 1)
     return n;
  else {
     int f;
     int pprev = 0;
     int prev = 1;

     for (int i = 2; i <= n; i++) {
        f = prev + pprev;
        pprev = prev;
        prev = f;
     }
     return f;
  }
}
```

n	Récursive	Itérative
10	0s	0s
20	0s	0s
40	1s	0s
45	8s	0s
50	87s	0s
60	3h	0,001s
100	77k ans	0,001s

Plan

Fonction puissance : itérativement

On souhaite calculer a^x, avec $a \in \mathbb{R}$ et x entier ≥ 1.

Version itérative

```
float pow_iter(float a, int x) {
    float res = a;
    for (int i = 1; i < x; i++)
        res = res * a;
    return res;
}
```

Nombre de multiplications nécessaires : $x - 1$

Fonction puissance : récursivement (1/2)

Une première formulation récursive :

$$a^x = \begin{cases} a & \text{si } x = 1 \\ a \cdot a^{x-1} & \text{si } x > 1 \end{cases}$$

```
float pow_rec1(float a, int x) {
    if (x == 1)
       return a;
    return a*pow_rec1(a, x-1);
}
```

Nombre de multiplications nécessaires : $x - 1$

Cette version est une simple réécriture de la version itérative.

Peut-on faire mieux ?

Fonction puissance : récursivement (2/2)

Une deuxième formulation récursive :

$$a^x = \begin{cases} a & \text{si } x = 1 \\ (a \cdot a)^{x/2} & \text{si } x > 1 \text{ et pair} \\ a \cdot (a \cdot a)^{(x-1)/2} & \text{si } x > 1 \text{ et impair} \end{cases}$$

```
float pow_rec2(float a, int x) {
    if (x == 1)
       return a;
    if (x % 2 == 0) // x pair
       return pow_rec2(a * a, x/2);
    else            // x impair
       return a * pow_rec2(a * a, (x-1)/2);
}
```

Nombre de multiplications nécessaires : entre $\log_2(x)$ et $2\log_2(x)$

- $x = 128 \Rightarrow 7$ multiplications au lieu de 127.

Plan

Une expérience

Y-a-t'il une différence lors de l'exécution de ces deux codes ?

```c
int fact_rec(int n) {
    return n*fact_rec(n-1);
}
int main() {
    fact_rec(100);
    return 0;
}
```

```c
int fact_iter(int n) {
    int res = 1;
    for (;; n--)
        res = res*n;
}
int main() {
    fact_iter(100);
    return 0;
}
```

Contexte d'appels de fonctions

A chaque appel de fonction, on doit retenir en mémoire (dans une zone appelée le Stack) le contexte de l'appel, c'est-à-dire :

- L'endroit où le code appelant doit continuer son exécution à la fin de l'appel
- La valeur des arguments fournis à la fonction
- La valeur des variables locales à la fonction

Cette information ne peut être supprimée qu'une fois l'appel terminé.

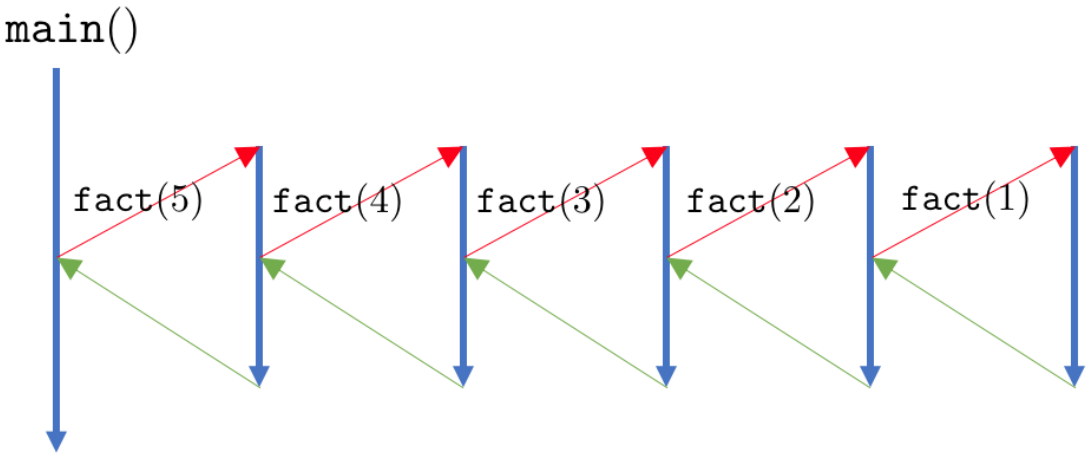
Dans le cas d'une fonction récursive, le nombre d'appels de fonction actifs à un moment donné peut être très important.

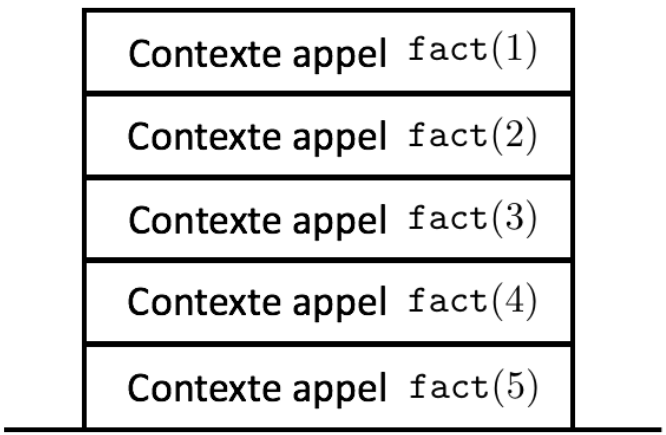
La récursivité a donc un coût mémoire dont il faut tenir compte.

Ce coût mémoire est directement proportionnel à la profondeur de l'arbre des appels récursifs.

Contexte d'appels de fonctions : illustration

```c
int fact(int n) {
    if (n == 1)
      return 1;
    return n*fact(n-1);
}
int main() {
    fact(5);
    return 0;
}
```





Une expérience : conclusion

Y-a-t'il une différence lors de l'exécution de ces deux codes ?

```c
int fact_rec(int n) {
   return n*fact_rec(n-1);
}
int main() {
   fact_rec(100);
   return 0;
}
```

⇒ Le programme s'arrête lorsque la mémoire est remplie.

```c
int fact_iter(int n) {
   int res = 1;
   for (;; n--)
      res = res*n;
}
int main() {
   fact_iter(100);
   return 0;
}
```

⇒ Le programme ne s'arrête jamais.

Synthèse

- Principal intérêt de la récursivité : élegance, simplicité, et lisibilité du code.
- Moins sujet à des bugs et analyse de correction facilitée par rapport aux boucles.
- Beaucoup d'algorithmes efficaces sont basés sur la récursivité (cf. Partie 5 et INFO0902).
- Mais la récursivité peut parfois mener à des solutions moins efficaces, voire très inefficaces.
- L'implémentation a un coût non négligeable en terme d'espace mémoire