Partie 2

Construction d'algorithmes

24 septembre 2019

Plan

1. Introduction
2. Construction d'algorithmes itératifs
3. Construction d'algorithmes récursifs

Construction de programme

Pour résoudre un problème de programmation complexe, on le découpe généralement en sous-problèmes plus simples à appréhender

- *Exemples de sous-problèmes pour afficher l'ensemble de Mandelbrot :*
  - *SP1 : calculer le module d'un nombre complexe,*
  - *SP2 : vérifier l'appartenance d'un point à l'ensemble,*
  - *SP3 : produire l'image en parcourant le plan complexe.*

Ces problèmes dépendent généralement les uns des autres.

- *Dans l'exemple, SP3 dépend de SP2 qui dépend de SP1.*

Avantages d'un découpage :

- Facilite l'implémentation : on peut résoudre chaque sous-problème indépendamment,
- Généralité : on peut partager du code entre différents programmes,
- Lisibilité et facilité de maintenance du code.

Construction de programmes

Résoudre un sous-problème élémentaire :

- Certains sont triviaux et requièrent d'établir une séquence simple d'instructions (p.ex., calculer le module d'un nombre complexe)
- D'autres sont plus complexes et nécessitent de répéter une séquence d'instructions selon un schéma dépendant des données (p.ex., déterminer l'appartenance à l'ensemble de Mandelbrot)

Deux grands types de solutions algorithmiques pour ces derniers cas :

- Solutions **itératives**, basées sur des boucles
- Solutions **récursives**, basées sur des fonctions qui s'invoquent elles-mêmes

On va (re)voir dans cette partie quelques grands principes pour la conception de ces deux types de solutions.

Plan

1. Introduction

2. Construction d'algorithmes itératifs
   - Technique de l'invariant
   - Illustrations
   - Correction d'algorithmes itératifs

3. Construction d'algorithmes récursifs

Construction d'algorithmes itératifs

Concevoir un algorithme itératif (correct) peut être un exercice très compliqué, surtout si on cherche une solution efficace.

Deux difficultés principales :

- Imaginer le schéma itératif permettant de résoudre le problème.
- Générer le code implémentant ce schéma en évitant les bugs.

Le premier problème est de loin le plus compliqué et cette compétence s'acquière quasi uniquement via la pratique (cf. INFO0902 pour des techniques génériques néanmoins).

La technique de l'invariant permet d'aborder formellement le second problème une fois le schéma de la boucle imaginé.

Invariant de boucle

Un **invariant de boucle** est une propriété définie sur les variables du programme qui définit précisément ce qui doit être calculé à chaque itération pour arriver au résultat escompté. Il résume l'état courant des calculs.

Identifier l'invariant revient à imaginer le schéma itératif de résolution du problème et est parfois non trivial.

Une fois l'invariant établi, implémenter la boucle peut par contre se faire de manière relativement automatique.

L'utilisation de l'invariant permet donc d'éviter les erreurs d'implémentation.

Invariant de boucle

Une assertion est une relation entre les variables et les données utilisées par le programme qui est vraie à un moment donné lors de l'exécution du programme.

Deux assertions particulières :

- Pré-condition P : condition que doivent remplir les entrées valides du programme
- Post-condition Q : condition qui exprime que le résultat du programme est celui attendu.

On cherche donc à écrire un programme, noté S, dont l'exécution dans tous les cas où P est vraie mène à ce que Q soit toujours vraie.

Lorsque c'est le cas, on dira que le triplet $\{P\}S\{Q\}$ est correct.

Exemple : Si $P = \{x \geq 0\}$ et $Q = \{y^2 = x\}$, le code $S = "y = \text{sqrt}(x);"$ rend le triplet $\{P\}S\{Q\}$ correct.

Invariant de boucle : plus formellement

```
{P}
INIT
while (B)
    CORPS
FIN
{Q}
```

```
{P}
INIT
{I}
while (B)
    {I et B} CORPS {I}
{I et non B}
FIN
{Q}
```

Dans le cas où le programme nécessite une boucle :

- On met en évidence une assertion particulière I, l'invariant de boucle, qui décrit l'état du programme pendant la boucle.
- On détermine ensuite le gardien B et les codes *INIT*, *CORPS* et *FIN* tels que les trois triplets suivants soient corrects :
  - $\{P\}$ INIT $\{I\}$
  - $\{I$ et $B\}$ CORPS $\{I\}$
  - $\{I$ et non $B\}$ FIN $\{Q\}$

(Si on a plusieurs boucles imbriquées, on les traite séparément.)

Ecriture du code sur base de l'invariant

Trois parties de code à écrire en se basant sur l'invariant :

1. Initialisation : $\{P\}$ INIT $\{I\}$
   - INIT doit rendre l'invariant vrai avant de rentrer dans la boucle et en partant de la pré-condition.
   - L'invariant identifie les variables nécessaires et comment les initialiser.
2. Maintenance : $\{I \text{ et } B\}$ CORPS $\{I\}$
   - CORPS doit faire avancer le problème en maintenant l'invariant en supposant que le gardien soit vrai.
   - L'invariant définit le schéma de la boucle.
3. Terminaison : $\{I \text{ et non } B\}$ FIN $\{Q\}$
   - FIN doit rendre la post-condition vraie en supposant que l'invariant est vérifié et le gardien est faux.
   - L'invariant définit comment finalement résoudre le problème.

Détermination de l'invariant

Pas de recette miracle pour déterminer l'invariant (ou de manière équivalente le schéma d'une boucle).

Quelques trucs néanmoins :

- Enlever une partie de la post-condition
- Remplacer dans la post-condition une constante par une variable
- Combiner les pré-conditions et post-conditions
- Raisonner par induction (voir plus loin)

Dans tous les cas, l'invariant doit faire apparaître toutes les variables du programme.

Pour un même problème, plusieurs invariants (et/ou gardiens de boucles) sont possibles qui mèneront à différentes implémentations de la boucle.

Illustration 1 : appartenance à l'ensemble de Mandelbrot

Pré et post-conditions :

- $P = \{\texttt{cr} \in \mathbb{R}, \texttt{ci} \in \mathbb{R}\}$
- $Q = \{\texttt{r} = 1 \text{ si } \forall n, 0 \leq n \leq N : |z_n| \leq 2, 0 \text{ sinon}\}$

où *cr* et *ci* sont les entrées du programme, *r* le résultat, *N* une constante, et z_n est la *n*-ème valeur de la suite définie précédemment avec $c \in \mathbb{C}$ tel que $c = \texttt{cr} + i\texttt{ci}$.

Schéma de la boucle :

- On calcule z_n pour des valeurs croissantes de *n* allant de 0 à *N*.
- A chaque itération, on calcule z_n à partir de la valeur z_{n-1} calculée à l'itération précédente.
- On s'arrête dès que $|z_n| > 2$.

Illustration 1 : appartenance à l'ensemble de Mandelbrot

Invariant :

$$I = \{(\forall n' : 0 \leq n' < \mathtt{n} : |z_{n'}| \leq 2) \text{ et } (\mathtt{zr} + i\mathtt{zi} = z_{\mathtt{n}}) \text{ et } (0 \leq \mathtt{n} \leq N)\},$$

où `n` est le compteur de boucle et `zr` et `zi` sont deux variables qui contiendront les résultats intermédiaires.

$$\overbrace{z_0, z_1, \ldots, z_{\mathtt{n}-1}}^{|\cdots| \leq 2}, \overbrace{\underbrace{z_{\mathtt{n}}}_{\mathtt{zr}+i\mathtt{zi}=z_{\mathtt{n}}}, z_{\mathtt{n}+1}, \ldots, z_N}^{|\cdots|?}$$

Gardien :

$$B = \{\mathtt{n} < N, \mathtt{zr}^2 + \mathtt{zi}^2 \leq 4\}.$$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{\mathtt{n}-1}}^{|\cdots|\leq 2}, \overbrace{\underbrace{z_{\mathtt{n}}}_{\mathtt{zr}+i\mathtt{zi}=z_{\mathtt{n}}}, z_{\mathtt{n}+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

INIT

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

CORPS

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

FIN

$\{Q\}$ $\{\mathtt{r} = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{\mathtt{n}-1}}^{|\cdots| \leq 2}, \overbrace{\underbrace{z_{\mathtt{n}}}_{\mathtt{zr}+i\mathtt{zi}=z_{\mathtt{n}}}, z_{\mathtt{n}+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;
int n = 0;
```

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$
CORPS
$\{I\}$

```
}
```

$\{I \text{ et non } B\}$
FIN
$\{Q\}$ $\{\mathtt{r} = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{n-1}}^{|\cdots|\leq 2}, \overbrace{\underbrace{z_n}_{zr+i\,zi=z_n}, z_{n+1}, \ldots, z_N}^{|\cdots|?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;
int n = 0;
```

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

```
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
```

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

FIN

$\{Q\}$ $\{r = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

$$\overbrace{z_0, z_1, \ldots, z_{n-1}}^{|\cdots| \leq 2}, \overbrace{\underbrace{z_n}_{zr + i zi = z_n}, z_{n+1}, \ldots, z_N}^{|\cdots| ?}$$

$\{P\}$ $\{cr \in \mathbb{R}, ci \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;
int n = 0;
```

$\{I\}$

```
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
```

$\{I \text{ et } B\}$

```
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
```

$\{I\}$

```
}
```

$\{I \text{ et non } B\}$

```
int r = (zr*zr+zi*zi <= 4.0);
```

$\{Q\}$ $\{r = 1 \text{ si } \exists n : 0 \leq n \leq N : |z_n| > 2, 0 \text{ sinon}\}$

Illustration 1 : appartenance à l'ensemble de Mandelbrot

```
double zr = 0;
double zi = 0;
int n = 0;
while((n < N) && (zr*zr + zi*zi <= 4.0)) {
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
  n++;
}
int r = (zr*zr+zi*zi <= 4.0);
```

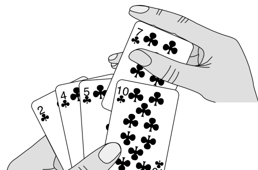
Illustration 2 : tri par insertion

On souhaite écrire une fonction pour trier un tableau A de valeurs entières. Le tri doit être effectué dans le tableau lui-même, via échanges d'éléments.

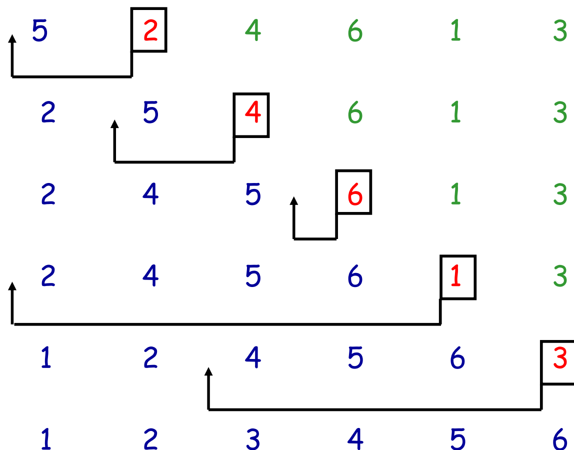
```
void insertion_sort(int A[], int N);
```

Principe de l'algorithme :

- On parcourt le tableau de gauche à droite en triant successivement les préfixes du tableau de tailles 2, 3, . . ., N.
- A chaque itération, on augmente la taille du préfixe trié en insérant le nouvel élément $A[i]$ à sa position dans le sous-tableau $A[0\,..\,i-1]$ précédemment ordonné.



Tri par insertion : graphiquement

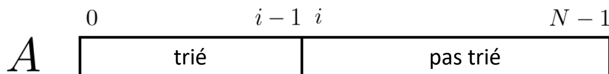


Tri par insertion : boucle externe

Pré-condition P_e : A est un tableau d'entiers de taille N.

Post-condition Q_e : A contient les éléments du tableau de départ triés par ordre croissant.

Invariant I_e (de la boucle externe) : Le sous-tableau $A[0 \,..\, i-1]$, avec $1 \leq i \leq N$, contient les i premiers éléments du tableau initial triés par ordre croissant.



Gardien B_e : $i < N$.

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

INITe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

FINe

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

```
int i = 1;
```

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

FINe

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

```
int i = 1;
```

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

CORPSe

$\{I_e\}$ $\{A[0\,..\,i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0\,..\,i-1]$ trié et $i = N\}$

```
-
```

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle externe

$\{P_e\}$ $\{A$ est un tableau d'entiers de taille $N\}$

```
int i = 1;
```

$\{I_e\}$ $\{A[0..i-1]$ trié$\}$

```
while (i<N) {
```

$\{I_e$ et $B_e\}$ $\{A[0..i-1]$ trié et $i < N\}$

CORPSe'

$\{Q_i\}$ $\{A[0..i]$ trié$\}$

```
    i++;
```

$\{I_e\}$ $\{A[0..i-1]$ trié$\}$

```
}
```

$\{I_e$ et non $B\}$ $\{A[0..i-1]$ trié et $i = N\}$

```
-
```

$\{Q_e\}$ $\{A$ contient les éléments du tableau de départ triés$\}$

Tri par insertion : boucle interne (CORPSe')

Pré-condition P_i : $\{I_e \text{ et } B_e\} = \{A[0\,..\,i-1] \text{ trié et } i < N\}$

Post-condition Q_i : $I_e = \{A[0\,..\,i] \text{ trié}\}$

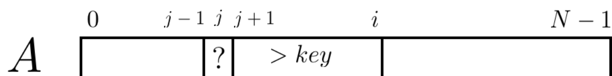
Idée de la boucle : Soit $key = A[i]$, la valeur à déplacer :

- On parcourt le sous-tableau $A[0\,..\,i-1]$ de droite à gauche.
- Tant que les éléments parcourus sont supérieurs à *key*, on les décale d'une position vers la droite.
- On insére *key* à la position finalement atteinte.

Tri par insertion : boucle interne

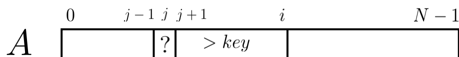
Invariant I_i : Soit j un nouvel indice et $key = A[i]$ la valeur à insérer :

- $A[0..j-1]$ et $A[j+1\,..\,i]$ sont triés par ordre croissant et ensemble contiennent tous les éléments du sous-tableau $A[0\,..\,i]$ initial, excepté key.
- $key < A[j+1]$



Gardien B_i : $\{j > 0$ et $A[j-1] > key\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

INITi

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

$\{I_i$ et $B_i\}$

CORPSi

$\{I_i\}$

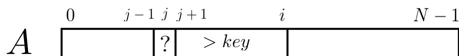
```
}
```

$\{I_i$ et non $B_i\}$

FINi

$\{Q_i\}$ $\{A[0\,..\,i-1]$ trié$\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

```
int key = A[i];
int j = i;
```

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

$\{I_i$ et $B_i\}$

CORPSi

$\{I_i\}$

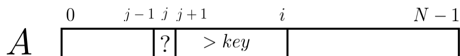
```
}
```

$\{I_i$ et non $B_i\}$

FINi

$\{Q_i\}$ $\{A[0\,..\,i-1]$ trié$\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0\,..\,i-1]$ trié et $i < N\}$

```
int key = A[i];
int j = i;
```

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

$\{I_i$ et $B_i\}$

```
  A[j] = A[j-1];
  j--;
```

$\{I_i\}$

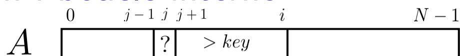
```
}
```

$\{I_i$ et non $B_i\}$

FINi

$\{Q_i\}$ $\{A[0\,..\,i-1]$ trié$\}$

Tri par insertion : boucle interne



$\{P_i\}$ $\{A[0\,..\,i-1] \text{ trié et } i < N\}$

```
int key = A[i];
int j = i;
```

$\{I_i\}$

```
while (j > 0 && A[j-1] > key) {
```

$\{I_i \text{ et } B_i\}$

```
  A[j] = A[j-1];
  j--;
```

$\{I_i\}$

```
}
```

$\{I_i \text{ et non } B_i\}$

```
A[j] = key;
i++;
```

$\{Q_i\}$ $\{A[0\,..\,i-1] \text{ trié}\}$

Tri par insertion : code complet

```
void insertion_sort(int A[], int N) {
  int i = 1;
  while (i < N) {
    int key = A[i];
    int j = i;
    while (j > 0 && A[j-1]>key) {
      A[j] = A[j-1];
      j--;
    }
    A[j] = key;
    i++;
  }
}
```

Synthèse

- L'expression d'un invariant permet de limiter les erreurs lors de l'implémentation d'une boucle.
- Vous devez prendre l'habitude d'exprimer l'invariant de toutes vos boucles **avant** leur implémentation, au minimum de manière informelle ou graphique.
- Dans la suite du cours, on fournira ponctuellement les invariants des boucles les plus compliquées.

Plan

1. Introduction

2. Construction d'algorithmes itératifs
   - Technique de l'invariant
   - Illustrations
   - Correction d'algorithmes itératifs

3. Construction d'algorithmes récursifs

Preuve de correction d'algorithmes itératifs

- Dans certains contextes, il est crucial de prouver formellement qu'un programme est correct (p.ex. dans le domaine médical ou en aéronautique).
- On peut toujours tester son programme empiriquement mais en général, il est impossible de considérer tous les cas possibles d'utilisation d'un code.

  Testing can only show the presence of bugs, not their absence

  E.W. Dijkstra

- L'analyse de correction de triplets et la technique de l'invariant de boucle peuvent aussi être utilisées pour prouver formellement qu'un algorithme itératif est correct (voir INFO2009).
- On peut automatiser une grosse partie de ces analyses, mais pas la dérivation des invariants de boucle.

Terminaison de boucle

Prouver qu'un triplet $\{P\}S\{Q\}$ est correct n'est pas suffisant dans le cas d'une boucle.

Il faut encore prouver que la boucle se termine.

Exemple : On peut prouver que le triplet ci-dessous est correct mais la boucle ne se termine pas toujours. On dira que le code est partiellement correct.

$\{\texttt{cr} \in \mathbb{R}, \texttt{ci} \in \mathbb{R}\}$

```
double zr = 0;
double zi = 0;

while(zr*zr + zi*zi <= 4.0)) {
  double temp;
  temp = zr*zr - zi*zi + cr;
  zi = 2*zr*zi + ci;
  zr = temp;
}
int r = (zr*zr+zi*zi <= 4.0);
```

$\{\texttt{r} = 1 \text{ si } \forall n \geq 0 : |z_n| \leq 2, 0 \text{ sinon}\}$

Terminaison de boucle

Pour prouver qu'une boucle se termine, on cherche une fonction de terminaison f :

- définie sur base des variables de l'algorithme et à valeur entière naturelle (≥ 0)
- telle que f décroît strictement suite à l'exécution du corps de la boucle
- telle que B implique $f > 0$

Puisque f décroit strictement, elle finira par atteindre 0 et donc à infirmer B.

Exemple :

- Mandelbrot : $f = N - n$
- Tri par insertion (boucle externe) : $f = N - i$

Terminaison de boucle

Il n'est pas toujours trivial de prouver la terminaison d'une boucle.

Personne n'a pu encore prouvé que la boucle suivante se terminait pour tout $n > 1$, bien qu'on l'ait prouvé empiriquement pour toutes les valeurs de $N < 1,25.2^{62}$.

```
void Algo(int n) {
  while(n != 1) {
    if (n % 2) // n est impair
      n = 3*n+1;
    else       // n est pair
      n = n/2;
  }
}
```

https://fr.wikipedia.org/wiki/Conjecture_de_Syracuse

Plan

1. Introduction

2. Construction d'algorithmes itératifs

3. Construction d'algorithmes récursifs
   - Principe
   - Illustrations
   - Implémentation de la récursivité

Algorithme récursif

- 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 récursives.
- Forme générale d'une fonction récursive (directe) :

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

Exemple 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);
}
```

Exemple 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
```

Exemple 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 *?*

Principe de construction d'un algorithme récursif

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

  Factorielle : n, le nombre dont on veut calculer la factorielle.
- Trouver une solution pour le(s) cas de base, c'est-à-dire des problèmes de petites tailles (la plupart du temps trivial).

  Factorielle : $n! = 1$ si $n \leq 1$.
- Trouver comment réduire le problème à un ou plusieurs sous-problème de tailles strictement plus petites.

  Factorielle : $n! = n * (n-1)!$ si $n > 1$

La dernière étape est la plus délicate. Equivalent à trouver l'invariant dans le cas d'algorithmes itératifs.

Plan

1. Introduction

2. Construction d'algorithmes itératifs

3. Construction d'algorithmes récursifs
   - Principe
   - Illustrations
   - Implémentation de la récursivité

Fonction puissance : solution itérative

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;
}
```

(Invariant ?)

Nombre de multiplications nécessaires : $x - 1$

Fonction puissance : récursivement (1/2)

Une première formulation récursive :

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

```c
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 :

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

```c
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.

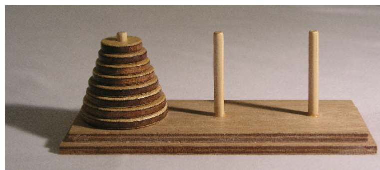
Fonction puissance : remarque

La solution précédente peut aussi s'implémenter de manière itérative.

Mais l'invariant de la boucle est obtenu sur base de la même formulation récursive et l'implémentation serait (un peu) moins immédiate.

Il existe des problèmes pour lesquels une solution itérative serait nettement plus complexe à implémenter qu'une solution récursive.

Le problème des tours de Hanoï cf. INFO2009



Source : wikipedia

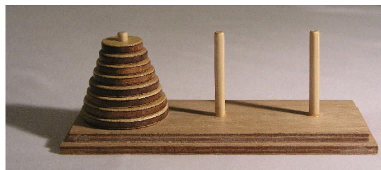
Objectif : transférer les n disques de la tour 1 à la tour 2, sans jamais mettre un disque sur un plus petit.

Solution récursive : Pour déplacer n disques :

- On déplace les $n-1$ disques du dessus vers la tour 3.
- On déplace le n-ème disque de la tour 1 vers la tour 2.
- On déplace les $n-1$ disques de la tour 3 vers la tour 2.

Solution itérative ?

Le problème des tours de Hanoï cf. INFO2009



Source : wikipedia

En fait, une solution itérative simple existe :

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

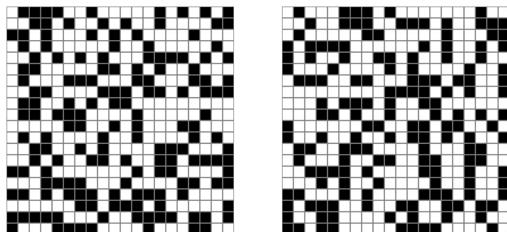
- *Bouger le plus petit disque à droite* ($1 \rightarrow 2$, $2 \rightarrow 3$ *ou* $3 \rightarrow 1$)
- *Bouger le seul autre disque qui peut être bougé*

Cette solution bien que strictement identique à la solution récursive est beaucoup plus compliquée à concevoir à partir de rien.

Son implémentation est aussi plus complexe car elle demande de représenter explicitement l'état des trois tours à chaque itération.

Percolation

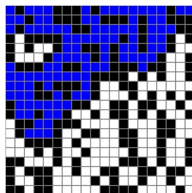
(Sedgewick et Wayne, 2016)

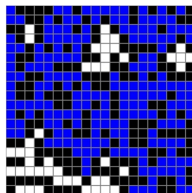


Problème : déterminer si de l'eau peut s'écouler dans un matériau poreux.

- Entrée : un tableau `grid` de taille $n \times n$, tel que $\texttt{grid}[i][j] = 0$ si le matériau est vide à la position (i, j), 1 si le matériau est plein.
- Sortie : vrai s'il existe un chemin partant de la première ligne, arrivant à la dernière ligne de la matrice et passant uniquement par des cases vides, faux sinon.

Percolation : découpage en sous-problème





Idée de la solution :

- Marquer chaque case de la grille pour laquelle il existe un chemin depuis une case vide de la première ligne.
- Vérifier qu'une case de la dernière ligne au moins a été marquée.

```c
// Valeur 2 utilisée comme marquage
int percolate(int **grid, int n) {
  flow(grid,n);   // fonction qui effectue le marquage des cases
  for (int j = 0; j < n; j++)
    if (grid[n-1][j] == 2)
      return 1;
  return 0;
}
```

Fonction `flow` : découpage en sous-problèmes

On définit une fonction `flowrec` :

```
void flowrec(int **M, int **marked, int N, int i, int j);
```

qui va marquer toutes les cases accessibles à partir de la position (i, j).

La fonction `flow` consiste alors simplement à appliquer `flowrec` à toutes les cases de la première ligne du tableau.

```
void flow(int **grid, int n) {
  for (int j = 0; j < n; j++)
     flowrec(grid, n, 0, j);
}
```

Une solution récursive pour `flowrec`

```c
void flowrec(int **grid, int n, int i, int j);
```

Cas de base : Ne rien faire si

- $\texttt{i} < 0$, $\texttt{i} \geq \texttt{n}$, $\texttt{j} < 0$, ou $\texttt{j} \geq \texttt{n}$,
  (la case est hors de la grille)
- $\texttt{grid}[\texttt{i}][\texttt{j}] == 1$,
  (la case est pleine)
- $\texttt{grid}[\texttt{i}][\texttt{j}] == 2$
  (la case a déjà été visitée lors d'un précédent appel récursif).

Cas inductif : Si on est pas dans un cas de base :

- On marque la case (i, j) qui est accessible.
- On applique récursivement la fonction `flowrec` à toutes les cases adjacentes à la case (i, j).

Une solution récursive pour `flowrec`

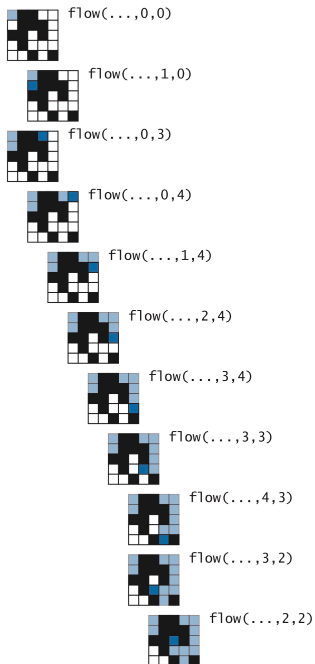
```c
void flow(int **grid, int n) {
  for (int j = 0; j < n; j++)
     flowrec(grid, n, 0, j);
}

void flowrec(int **grid, int n, int i, int j) {
  // Cas de base
  if (i < 0 || i >= n || j < 0 || j >= n) return;
  if (grid[i][j] == 1 || grid[i][j] == 2) return;
  // Cas inductif
  grid[i][j] = 2;
  flowrec(grid, n, i+1, j); // Bas
  flowrec(grid, n, i, j+1); // Droite
  flowrec(grid, n, i, j-1); // Gauche
  flowrec(grid, n, i-1, j); // Haut
}
```

Correction ? Complexité ?

Note : `flowrec` implémente ce qu'on appelle une recherche en profondeur d'abord (*depth-first search).*

Illustration

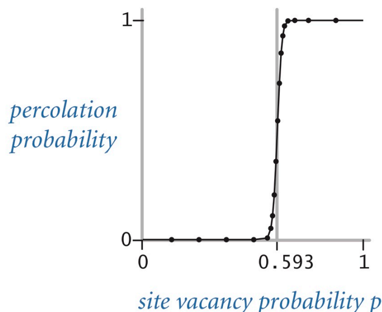


(Sedgewick et Wayne, 2016)

Application : simulations numériques

Cette fonction permet par exemple d'étudier l'impact de la porosité du matériau sur la probabilité de percolation.

Pour des grilles de taille 100x100 (10000 grilles aléatoires générées par point) :



(Sedgewick et Wayne, 2016)

Efficacité de codes récursifs

Dans le cas de la fonction puissance, des tours de Hanoï et de la percolation, la solution récursive donne un code optimal en terme de complexité.

Dans certains cas, l'utilisation naïve de la récursivité peut cependant mener à une solution très sous-optimale.

Exemple : calcul des nombres de Fibonacci

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

Nombres de Fibonacci : implémentation récursive

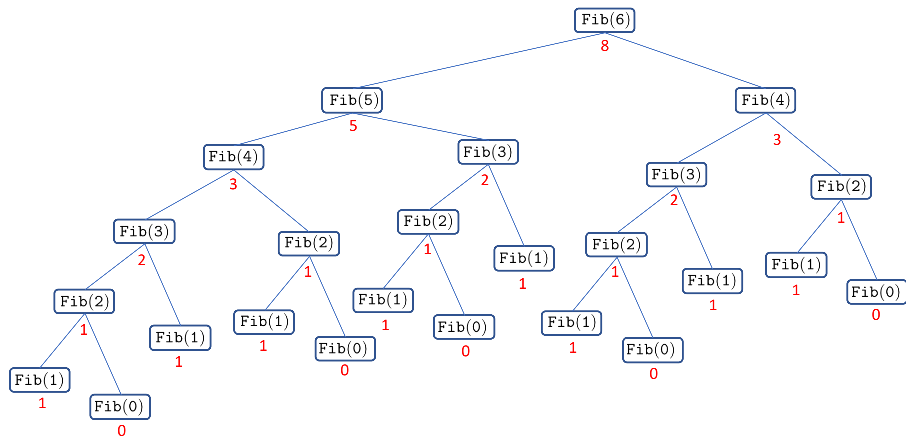
Implémentation récursive directe :

```
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 :

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

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

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 $n \leq 1$, $\texttt{fact}(n)$ 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 $n * \texttt{fact}(\texttt{n} - 1)$, 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

1. Introduction

2. Construction d'algorithmes itératifs
   - Technique de l'invariant
   - Illustrations
   - Correction d'algorithmes itératifs

3. Construction d'algorithmes récursifs
   - Principe
   - Illustrations
   - Implémentation de la récursivité

Une expérience

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

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

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

Contextes 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.

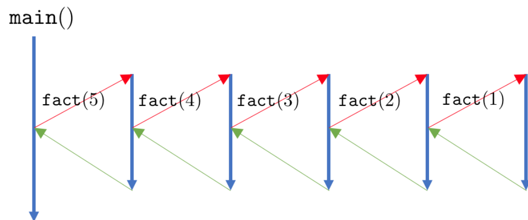
Dans le cas d'une fonction récursive, le nombre d'appels de fonction actifs à un moment donné peut être très important.

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.

Contextes 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;
}
```



Contexte appel fact(1)

Contexte appel fact(2)

Contexte appel fact(3)

Contexte appel fact(4)

Contexte appel fact(5)

Une expérience : conclusion

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

```
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 qui lui est allouée est remplie.

```
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.

Récursivité terminale

Une fonction est récursive terminale s'il n'y a plus de calcul à effectuer une fois l'appel récursif terminé.

Exemple : le calcul du PGCD :

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

  if (b == 0)
    return a;

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

Il est dans ce cas inutile de stocker le contexte des appels.

Les compilateurs modernes sont capables de détecter ce type de récursion (et d'autres plus compliquées) et de ne pas utiliser de mémoire inutile.

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 l'utilisation naïve de la récursivité peut parfois mener à des solutions moins efficaces, voire très inefficaces.
- Il existe néanmoins des techniques systématiques pour améliorer l'efficacité d'une solution récursive (cf. INFO0902 et INFO0540).
- L'implémentation a souvent un coût non négligeable en terme d'espace mémoire dont il faut tenir compte.