1. Récursion

LE COURS nous place dans le domaine de connaissances sur **la conception, l'analyse et l'implantation d'algorithmes**. Mais tout d'abord, on a besoin d'un cadre formel pour les décrire.

1.1 Modèles de calcul

Un algorithme est la formalisation de la suite d'opérations à exécuter pour résoudre un problème bien défini. Il doit terminer en un temps fini, et fournir la réponse à partir des données de l'entrée. La description assume un modèle de calcul définissant les **instructions élémentaires** qui forment le vocabulaire à utiliser.

Machine de Turing. La **machine de Turing**[1] comprend seulement un ruban «magnétique», et une tête de lecture-écriture. L'algorithme sur la machine de Turing comprend un ensemble (fini) d'états et un ensemble de règles d'action pour la tête et les transitions entre états.

[1] W(fr):Machine de Turing

FIG. 1: Alan Turing (1912–1954)

Machine RAM. La gestion de mémoire est difficile avec une machine de Turing (la tête glisse par une case à la fois ⇒ accès séquentiel aux cellules). Dans une **machine RAM**[2] on peut adresser le mémoire directement, et y stocker les données. L'ensemble d'instructions imite les langages des CPUs contemporaines.

[2] W(fr):Machine RAM

Pseudocode et implémentation. En général, on va décrire nos algorithmes en **pseudocode**. Le syntaxe est laxiste et dépend de l'auteur, mais les instructions doivent être faciles à implanter en un langage de programmation. (Dans ce cours, je vais illustrer les implémentations largement en Java, et décrire les algorithmes en un mélange de notation mathématique, avec un vocabulaire d'instructions proche de Java/C/Ruby.) Souvent, on doit chercher un compromis entre l'algorithme conceptuel et les contraintes pratiques de l'implantation. Voir l'exemple de Fig. 2 où la version Java introduit le typage des éléments et une représentation spéciale pour ∞. En résultat, le code est moins générique que l'algorithme en pseudocode.

MIN-ITER $(x[0..n-1])$

M1 initialiser min $\leftarrow \infty$

M2 **for** $i \leftarrow 0, \ldots, n-1$ **do**

M3 **if** $x[i] <$ min **then** min $\leftarrow x[i]$

M4 **return** min

```
static int minIter(int[] x)
{
    int m = Integer.MAX_VALUE; // "infini"
    for (int i=0; i<x.length; i++)
        if (x[i]<m) m=x[i];
    return m;
}
```

FIG. 2: Pseudocode (style «pidgin Java») et implémentation pour un algorithme simple qui trouve le minimum dans un tableau. Nos idiosyncrasies de notation : indexage du tableau ($x[0..n-1]$: n éléments, premier à l'indice 0), affectation ($a \leftarrow b$), mots-clés en gras (**if**..**then**, **for**..**do**, **return**), tabulation pour imbriquer les blocs d'instruction.

1.2 Analyse d'algorithmes

L'analyse d'un algorithme caractérise le comportement en fonction de l'entrée et de la sortie. Le modèle de calcul assumé spécifie le coût (temps d'exécution) associé avec chaque instruction, et la taille (usage de mémoire) des données. Avec une machine de Turing, chaque instruction (=règle) a le même coût unitaire, et le temps de calcul se mesure par le total des actions (déplacements de la tête / transitions d'état).

modèle de calcul :
⋆ jeu d'instructions
⋆ coût (temps) des instructions
⋆ usage de mémoire

Quand on analyse l'efficacité d'un algorithme en pseudocode, on se sert des coûts variables qui correspondent à l'exécution après implémentation, en langage machine ou machine RAM. Par exemple, la machine RAM typique ne possède d'instruction pour boucler[3]. Le jeu d'instructions de processeurs inclut plutôt des branchements conditionnels dont on se sert pour implanter les boucles `for` ou `while`. Une itération de la boucle `for` dans un tel modèle comprend au minimum

- ⋆ la mise à jour de la variable d'itération : une affectation
- ⋆ l'évaluation de condition de terminaison : une comparaison
- ⋆ un saut selon le résultat

[3] Par contre, le langage machine `x86_64` inclut l'instruction `LOOP` qui décremente un compteur (régistre `CX`) et prend un saut s'il n'est pas égal à 0.

Exemple 1.1. On prend l'exemple de rechercher le minimum dans le tableau (MIN-ITER de Figure 2), implanté sur une machine RAM sans boucle[4]. Pour un tableau de taille n, l'exécution de la boucle **for** prend $(2n+1)$ comparaisons, et $(n+k+1)$ affectations où $k \leq n$ est le nombre de fois que $x[i]$ est plus petit que ces antécédants $x[i-1], \ldots, x[0]$. ♠

[4] MIN-ITER sur machine RAM sans boucles :

$m \leftarrow \infty\,;\, i \leftarrow 0$
S : **if** $i \geq n$ **then JUMP** E
if $x[i] < m$ **then** $m \leftarrow x[i]$
$i \leftarrow i+1\,;$ **JUMP** S
E : **return** m

1.3 Formules récursives

On examine ici deux fonctions définies par des expressions récursives simples : la factorielle et les nombres Fibonacci.

La factorielle

Définition 1.1. *On définit la* factorielle $n!$ *d'un nombre naturel* $n \in \{0,1,2,3\dots\}$ *par*

$$n! = \begin{cases} 1 & \{n=0\} \\ n\cdot(n-1)! & \{n>0\} \end{cases}$$

La factorielle existe pour tout entier $n \geq 0$:

$$0! = 1$$
$$n! = 1\times 2\times 3\times\cdots\times n = \prod_{k=1}^{n} k. \qquad \{n=1,2,3,\dots\}$$

La factorielle croît très rapidement — c'est une fonction **superexponentielle** : pour tout $c>1$ fixe, il existe un $n_0(c)$ t.q.

$$c^n < n! \qquad \{n = n_0(c), n_0(c)+1,\dots\} \tag{1.1}$$

Par exemple, avec $c=10$, $n_0(c)=25$ suffit :

$$25! = 15511210043330985984000000 > 10^{25}.$$

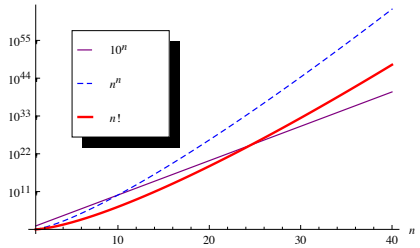

FIG. 3: Croissance superexponentielle de la factorielle

Théorème 1.1 (Formule de Stirling). *Pour tout* $n = 1,2,\dots,$

$$n! = \underbrace{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}_{\text{formule de Stirling}} \times \underbrace{\exp\left(\frac{\theta_n}{12n}\right)}_{\text{erreur de l'ordre } 1/n} \qquad \{0<\theta_n<1\} \tag{1.2}$$

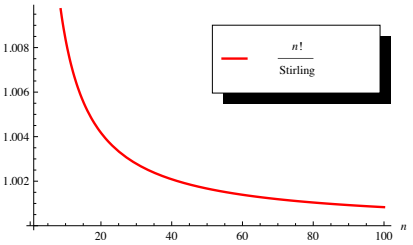

FIG. 4: La formule de Stirling.

On écrit[5] alors

$$n! \sim S(n) = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

La formule de Stirling donne une borne inférieure serrée :

$$e^{0/12n} = 1 < \frac{n!}{S(n)} = e^{\theta_n/12n} < \underbrace{e^{1/12n} < 1+\frac{1}{12n}}_{\text{Taylor : } e^x = 1+x+x^2/2!+\cdots}$$

[5] $f(n) \sim g(n)$ («f est asymptotiquement égale à g») ssi $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

Nombres Fibonacci

Définition 1.2. *On définit les* ***nombres Fibonacci***[6] $F(n)$ *pour* $n = 0,1,2,\dots$ *:*

$$F(0) = 0$$
$$F(1) = 1 \tag{1.3}$$
$$F(n) = F(n-1)+F(n-2) \quad \{n-2,3,4,\dots\}$$

FIG. 5: Leonardo Fibonacci (1175–1250)

[6] W(fr):Suite de Fibonacci

Théorème 1.2 (Formule de Binet)**.** *Soit*

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618\cdots \qquad \textit{et} \qquad \bar{\phi} = 1-\phi = \frac{1-\sqrt{5}}{2} = -0.618\cdots .$$

On a alors

$$F(n) = \frac{\phi^n - \bar{\phi}^n}{\sqrt{5}}. \tag{1.4}$$

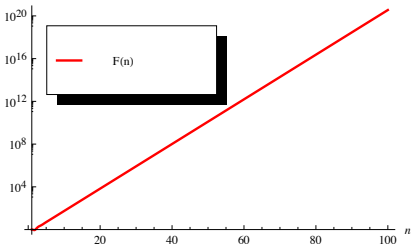

FIG. 6: Croissance exponentielle des nombres Fibonacci

Dans la formule (1.4), $|\bar{\phi}| < 1$ et donc $F(n)$ a une croissance exponentielle : $F(n) \sim \phi^n/\sqrt{5}$ (v. Fig. 6). En fait, comme $|\bar{\phi}^n/\sqrt{5}| < 1/2$ pour tout $n \geq 0$ et $F(n)$ est entier, on voit aussi que $F(n)$ égale à $\phi^n/\sqrt{5}$, arrondi au plus proche entier :

$$F(n) = \left\lfloor \phi^n/\sqrt{5} + 1/2 \right\rfloor.$$

Démonstration de 1.2. On définit la fonction $f(x) = \sum_{n=0}^{\infty} F(n)\cdot x^n$. Alors

$$\begin{aligned} f(x) &= 0 + x + \sum_{n=2}^{\infty} \big(F(n-1)+F(n-2)\big)x^n && \textit{(par Définition)} \\ &= x + x\cdot \underbrace{\sum_{n=2}^{\infty} F(n-1)x^{n-1}}_{=f(x)} + x^2 \underbrace{\sum_{n=2}^{\infty} F(n-2)x^{n-2}}_{=f(x)}, \end{aligned}$$

d'où

$$\begin{aligned} f(x) &= \frac{x}{1-x-x^2} = \frac{x}{(1-\phi x)(1-\bar{\phi}x)} = \frac{1/\sqrt{5}}{1-\phi x} - \frac{1/\sqrt{5}}{1-\bar{\phi}x} \\ &= \frac{1}{\sqrt{5}}\Big(\sum_{n=0}^{\infty}(\phi x)^n - \sum_{n=0}^{\infty}(\bar{\phi}x)^n\Big) && \textit{(somme géométrique)} \\ &= \sum_{n=0}^{\infty} \frac{\phi^n - \bar{\phi}^n}{\sqrt{5}}\cdot x^n. \end{aligned}$$

L'égalité vaut pour tout $|x| < 1/\phi$ ce qui n'est possible que si $F(n) = \frac{\phi^n - \bar{\phi}^n}{\sqrt{5}}$ dans tout terme $F(n)x^n$. ■

1.4 *Algorithmes récursifs*

Définition 1.1 se traduit en un **algorithme récursif** :

```
    FACT(n)                                  //(calcul de n!)
F1  if n = 0 then return 1
F2  else return n × FACT(n − 1)              // appel récursif
```

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

FIG. 7: Algorithme récursif pour la factorielle, et l'implémentation. Notons que $17! > 2^{31} - 1 =$ `Integer.MAX_VALUE`, donc la version Java aura des problèmes numériques avec $n \geq 17$: la valeur retournée est en fait l'entier signé $n! \bmod 2^{32}$.

On peut vérifier que l'algorithme satisfait les règles minimales pour récursion :

(1) il y a (au moins) un **cas terminal**, et

(2) chaque **appel récursif** nous rend «plus proche» à un cas terminal.

En conséquence, l'algorithme finit en un nombre fini d'appels récursifs pour tout n. (D'ailleurs ce n'est pas la meilleure façon de calculer $n!$ pour un grand n : on peut juste évaluer[7] la formule de Stirling en un temps constant pour tout n.)

[7] Au lieu de représenter $n!$ directement, on emploit plutôt la formule de Stirling pour calculer $\log(n!)$ comme nombre flottant.

Tours de Hanoï

La récursion peut nous fournir une solution simple à un problème difficile. On examine deux exemples : tours de Hanoî et l'algorithme d'Euclide.

Le but dans le jeu de réflexion *Tours de Hanoï*[8] est de déplacer des disques à diamètres différents $(1, 2, \ldots, n)$ d'une tour de départ à une tour d'arrivée en passant par une tour intermédiaire, tout en respectant Règles 1 et 2 ci-dessous. Opération $\text{MOVE}(i \to j)$ déplace le disque en haut de tour i à tour j. Les disques sont en ordre décroissant au début, et il y a trois tours.

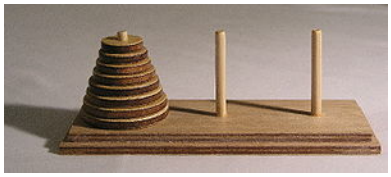

FIG. 8: Tours de Hanoï avec 8 disques.

[8] W(fr):Tours de Hanoï

Règle 1. On ne peut déplacer plus d'un disque à la fois. Un déplacement consiste de mettre le disque supérieur sur une tour au-dessus des autres disques (s'il y en a).

Règle 2. On ne peut placer un disque que sur un autre disque plus grand ou sur un emplacement vide.

FIG. 9: Édouard Lucas (1842–1891), inventeur du jeu

Une solution simple est fournie en définissant une procédure récursive $\text{HANOI}(i \curvearrowright k \curvearrowright j, n)$ qui déplace les n disques supérieurs sur tour i vers tour j en utilisant la tour intermédiare k.

```
    HANOI(i ⌒ j ⌒ k, n)
H1  if n ≠ 0 then
H2      HANOI(i ⌒ j ⌒ k, n − 1)
H3      MOVE(i → j)
H4      HANOI(k ⌒ i ⌒ j, n − 1)
```

Théorème 1.3. *La procédure* HANOI *performe* $2^n - 1$ *déplacements pour arranger* n *disques.*

Démonstration. Soit $D(n)$ le nombre de déplacements (faits par Ligne H3). Par inspection,

$$D(n) = \begin{cases} 0 & \{n = 0\} \\ 2 \cdot D(n-1) + 1 & \{n > 0\} \end{cases} \tag{1.5}$$

On définit $d(x) = \sum_{n=0}^{\infty} D(n)x^n$. Selon Eq. (1.5),

$$d(x) = 0 \cdot x^0 + \sum_{n=1}^{\infty} \big(2D(n-1)+1\big)x^n = 2x \underbrace{\sum_{n-1}^{\infty} D(n-1)x^{n-1}}_{=d(x)} + \sum_{n=1}^{\infty} x^n = 2xd(x) + \frac{x}{1-x}$$

$$d(x) = \frac{x/(1-x)}{1-2x} = \frac{1}{1-2x} - \frac{1}{1-x} = \sum_{n=0}^{\infty} (2x)^n - \sum_{n=0}^{\infty} x^n = \sum_{n=0}^{\infty} \underbrace{\big(2^n - 1\big)}_{D(n)} x^n.$$

■

Algorithme d'Euclide

L'algorithme d'Euclide[9] trouve le plus grand commun diviseur de deux entiers :

[9] W(fr):algorithme d'Euclide

```
GCD(a, b) // {b ≤ a}
E1  if b = 0 then return a
E2  return GCD(b, a mod b);
```

```
int gcd(int a, int b)
{   assert (b<=a && b>=0);
    if (b==0) return a;
    else return gcd(b, a%b);
}
```

FIG. 10: Portrait d'Euclide [d'Alexandrie] (vers 300 avant notre ère), pent par Joos van Wassenhove vers 1475

Théorème 1.4. *Pour tout $a \geq b \geq 0$, on définit*

$$N(a,b) = \begin{cases} 0 & \{b = 0\} \\ \max\{n \colon a \geq F(n+1), b \geq F(n)\} & \{b > 1\}. \end{cases}$$

Si $b > 0$, alors

$$N(b, a \bmod b) < N(a,b)$$

Démonstration. Soit $N(x) = \max\{n \colon F(n) \leq x < F(n+1)\}$. Par définition, $N(a,b) = \min\{N(a)-1, N(b)\}$ pour tout $a \geq b > 0$. On considère $N(a,b) = n$.

Cas 1 : $N(a) = N(b) = n+1$. Alors $a \bmod b = (a-b) \bmod b \leq a - b < F(n)$. On a $N(b, a \bmod b) = \min\{N(b)-1, N(a \bmod b)\} < n$.

Cas 2 : $N(b) = n, N(a) \geq n+1$. Alors $N(b, a \bmod b) = \min\{n-1, N(a \bmod b)\} < n$. ■

a	b	$N(a,b)$
	0	0
1	1	1
2, 3, …	1	2
2	2	2
3, 4, …	2	3
3, 4	3, 4	3
5, 6, …	3, 4	4
5, 6, 7	5, 6, 7	4
8, 9, …	5, 6, 7	5
…		

Par Théorème 1.4, $N(a,b)$ décroît lors de chaque appel récursif, et si $N(a,b) = 0$, il n'y plus d'appels. En conséquence, le nombre d'appels au total doit être $N(a,b) - 1$ au plus. Or, selon la formule de Binet, $F(n) \sim 5^{-1/2}\phi^n$, d'où $n'(b) = \left\lfloor \log_\phi\big(b\sqrt{5}\big) \right\rfloor$ nous donne une borne logarithmique en b sur le nombre d'appels. Ce qui est plus, la preuve nous donne deux joyaux au gratuit : (1) les nombres Fibonacci sont des nombres premiers entre eux, et (2) le pire cas de l'algorithme est un appel avec nombres Fibonacci consécutifs.