Introduction

Algorithms

Algorithmique

algorithm design

- ⋆ conception d'algorithmes
- ⋆ analyse d'algorithmes algorithm analysis
- ⋆ implémentation d'algorithmes implementation of algorithms

question 0 : comment décrire un algorithme ?

how to write an algorithm?

Algorithms

Algorithme

Algorithme = formalisation de la suite d'opérations à exécuter pour résoudre un problème bien défini

formalization of the sequence of operations to be performed to solve a finite problem

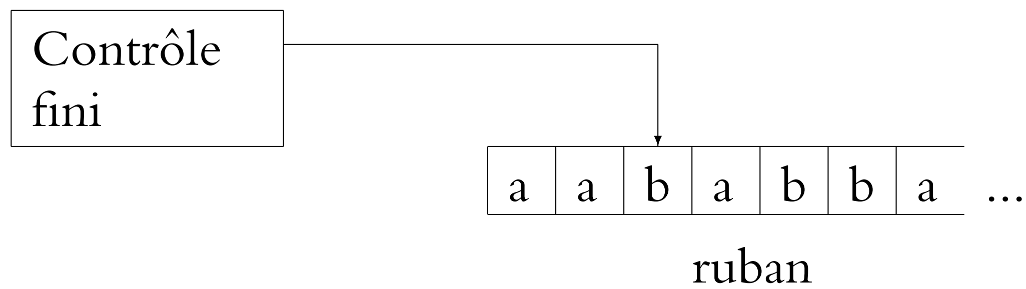
→ vocabulaire contrôlé (modèle de calcul, syntaxe d'instructions)

→ se termine en un temps fini

→ fournit la solution au problème

Machine de Turing

Modèle :



Caractéristiques :

– Une machine de Turing peut lire ou écrire
– La tête de lecture peut bouger à gauche ou à droite
– Le ruban est infini vers la droite
– Quand on décide d'accepter ou de rejeter, c'est une décision finale.

Machine de Turing

Une **machine de Turing (MT)** est un 7-tuplet $(Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ où

- Q est l'ensemble fini d'**états**,
- Σ est l'**alphabet**,
- Γ est l'**alphabet de ruban**,
  $\sqcup \in \Gamma$ et $\Sigma \subseteq \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
  est la **fonction de transition**,
- q_0 est l'état **initial**,
- q_a est l'état **acceptant**,
- q_r est l'état **rejetant**.

Thèse de Church-Turing : calculabilité = faisabilité sur machine de Turing

Machine RAM

abstraction de langage machine (instructions arithmétiques, et de contrôle)

accès direct au mémoire vive, variables

variable = abstraction d'un emplacement en mémoire (von Neumann)
nom + adresse (lvalue) + valeur (rvalue) + type + portée

Pseudocode

langage imitant de vrais langages de programmation de haut niveau

$\textsc{Min-iter}\big(x[0..n-1]\big)$

M1 initialiser $\mathsf{min} \leftarrow \infty$

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

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

M4 **return** min

implémentation «facile» en un langage de programmation choisi

```
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;
}
```

Compromis

```
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;
}
```

- ⋆ typage des variables `int` $\Rightarrow x[i] \in \left\{-2^{31}, -2^{31}+1, \ldots, 2^{31}-1\right\}$
- ⋆ représentation de ∞
- ⋆ exception possible quand `x` $==$ null

Analyse d'algorithmes

modèle de calcul définit le coût/temps d'exécution des instructions de base

usage de mémoire (représentation de types, mécanismes de gestion)

temps de calcul bien caractérisé par le nombre d'opérations *typiques* que l'algorithme exécute : fonction de la taille de l'entrée (et sortie)

Exemple : MIN-ITER fait $2n-1$ comparaisons pour trouver le minimum ($x[i] < m$, $i < n$) dans un tableau de longueur n

Récursion

Factorielle $n!$ (nombre de permutations sur n éléments distincts) :

$$0! = 1$$

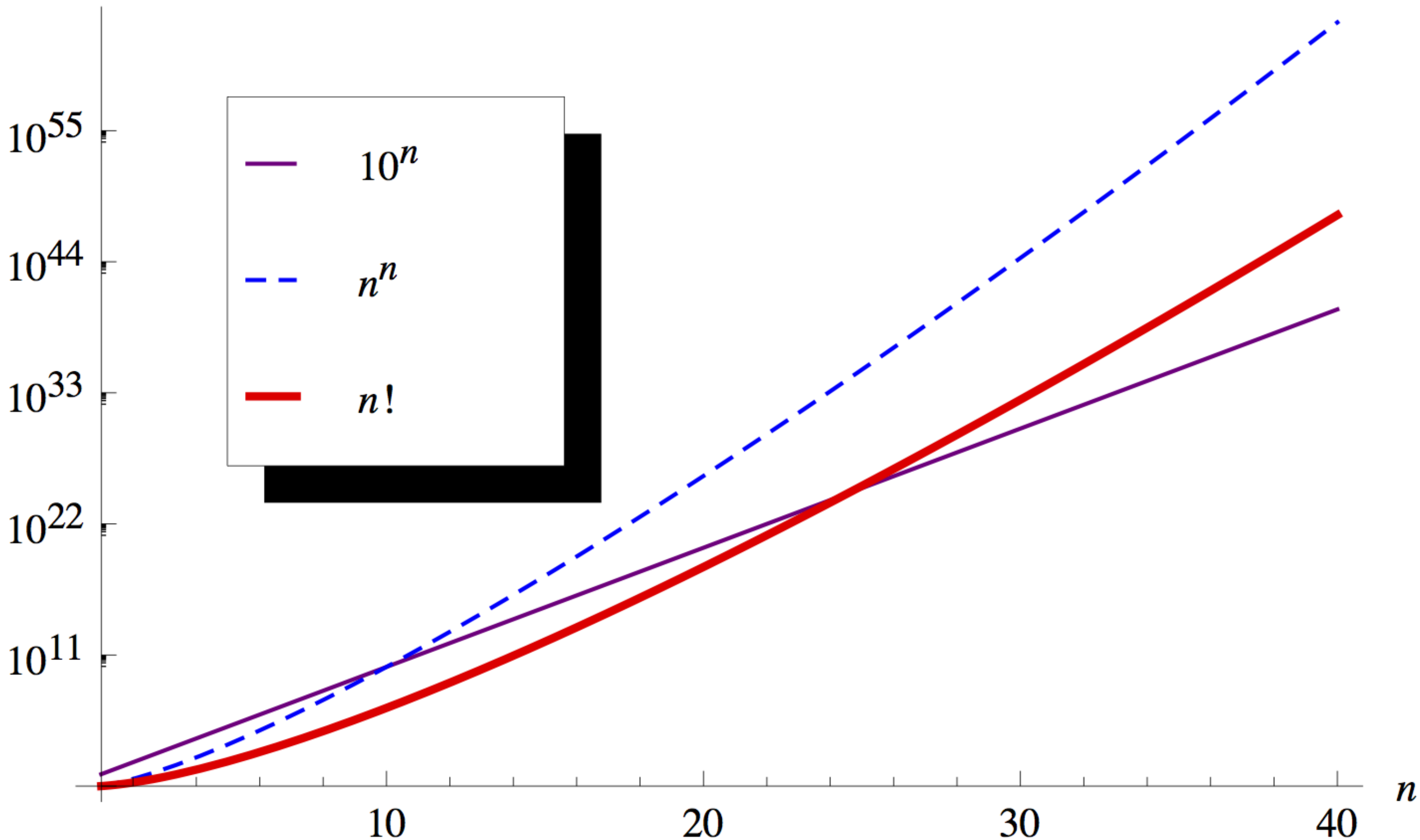
$$n! = 1 \times 2 \times 3 \times \cdots \times n = \prod_{k=1}^{n} k. \qquad \{n = 1, 2, 3, \ldots\}$$

Définition récursive (cas de base + cas récursif)

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

Croissance

Croissance superexponentielle (noter échelle logarithmique) :

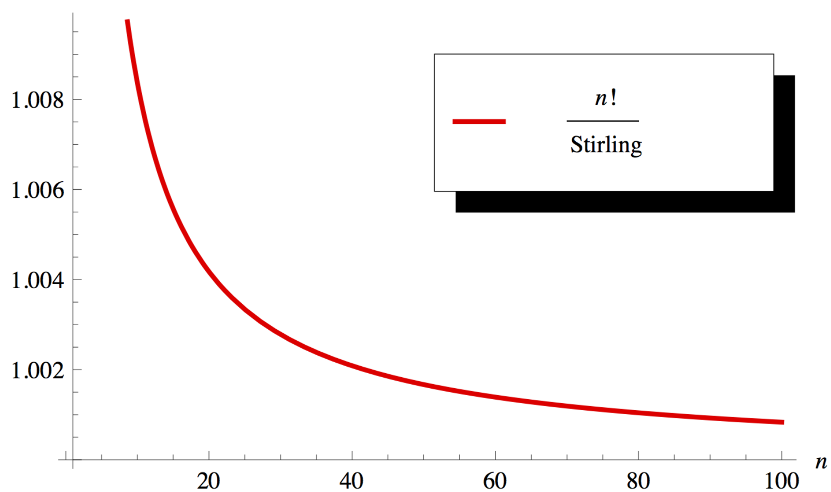


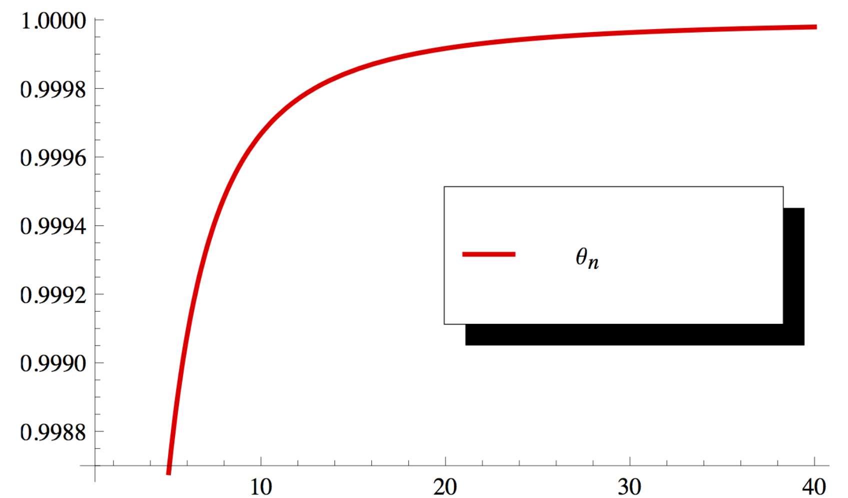
Formule de Stirling

Pour tout $n = 1, 2, \ldots,$

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

Très précis !





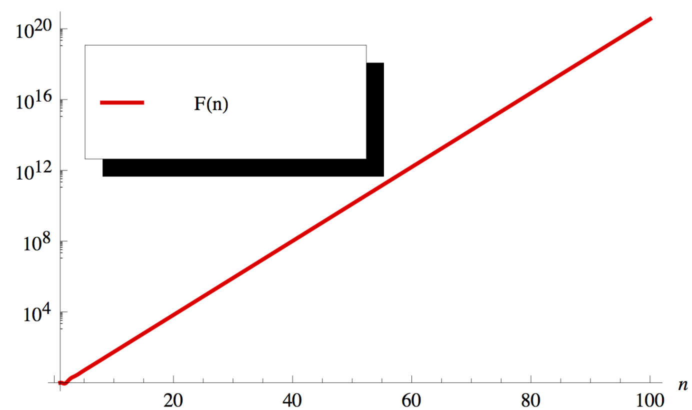
Nombres Fibonacci

Définition récursive (cas de base + cas récursif) :

$$
\begin{aligned}
F(0) &= 0 \\
F(1) &= 1 \\
F(n) &= F(n-1) + F(n-2) \;\; \{n - 2, 3, 4, \ldots\}
\end{aligned}
\tag{2}
$$

n	0	1	2	3	4	5	6	7	8	9	10	11
$F(n)$	0	1	1	2	3	5	8	13	21	34	55	89

Croissance exponentielle (sur lin-log) :



Formule de Binet

Soit $\phi = \frac{1+\sqrt{5}}{2} = 1.618\cdots$ [la proportion divine], et $\bar{\phi} = 1 - \phi = \frac{1-\sqrt{5}}{2} = -0.618\cdots$

Pour tout $n = 0, 1, 2, \ldots$, on a

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

Algorithmes récursifs

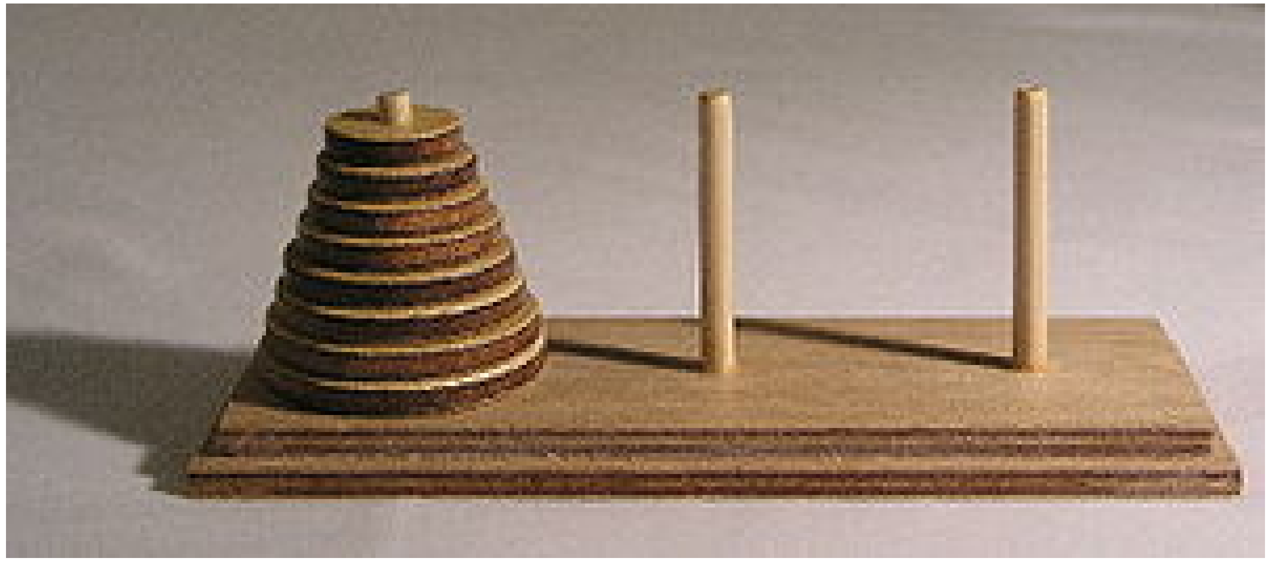
```
    MIN-RECUR(x[0..n-1], k)   // premier appel avec k = 0
M1  if k = n then return ∞
M2  m ← MIN-RECUR(x, k+1)   // appel récursif
M3  return min{x[k], m}
```

```
    FACT(n)                   //(calcul de n!)
F1  if n = 0 then return 1
F2  f ← FACT(n-1)             // appel récursif
F3  return n · f
```

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

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

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.

Tours de Hanoï — algo

$\textsc{Hanoi}(i \curvearrowright j \curvearrowright k, n)$

H1 **if** $n \neq 0$ **then**

H2 $\quad\textsc{Hanoi}(i \curvearrowright j \curvearrowright k, n-1)$

H3 $\quad\textsc{Move}(i \to j)$

H4 $\quad\textsc{Hanoi}(k \curvearrowright i \curvearrowright j, n-1)$

On mesure le temps de calcul par $D(n) =$ nombre de déplacements :

$$D(n) = \begin{cases} 0 & \{n = 0\} \\ 2 \cdot D(n-1) + 1 & \{n > 0\} \end{cases} \tag{3}$$

Thm : $D(n) = 2^n - 1$.

Algorithme d'Euclide

L'algorithme d'Euclides trouve le plus grand commun diviseur de deux entiers non-négatifs :

$\textsc{gcd}(a, b)$ // $\{b \leq a\}$

E1 **if** $b = 0$ **then return** a

E2 **return** $\textsc{gcd}\big(b, a \bmod b\big)$;

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

Algorithme d'Euclide — analyse

Définir

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

Lemme : Si $b > 0$, alors

$$N(b, a \mathbin{\mathsf{mod}} b) < N(a,b)$$

Thm : Si $N(a,b) = n$ lors de l'appel $\textsc{gcd}(a,b)$, alors l'algorithme finit en $(n-1)$ appels récursifs au plus.

Algorithme d'Euclide — conclusion

Trouvailles dans la preuve :

- ☞ les nombres Fibonacci sont premiers entre eux
- ☞ le pire cas de l'algorithme arrive quand les paramètres sont des nombres Fibonacci consécutifs