

Problèmes et Complexité

Où l'on parle de la prolifération des lapins

Jean-Marc.Vincent@univ-grenoble-alpes.fr

Laboratoire LIG, Équipe Inria POLARIS

DIU EIL
Grenoble 2019



- 1 **DIVERTISSEMENT : les lapins**
- 2 **CALCUL DE x^n : l'algorithme du débutant**
- 3 **COMPLEXITÉ d'un problème**

ALGORITHMIQUE ET MODÉLISATION

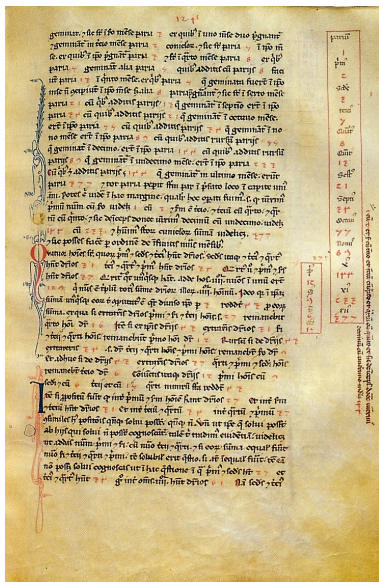
- 1 **DIVERTISSEMENT : les lapins**
- 2 CALCUL DE x^n : l'algorithme du débutant
- 3 COMPLEXITÉ d'un problème

LES LAPINS DE FIBONNACCI (1170-1250)

Leonardo Pisano dit Fibonnaci rédige le
Liber abaci vers 1202



Possédant initialement un couple de lapins, combien de couples obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ?



LES LAPINS DE FIBONNACCI (SUITE)

F_n = population de couples de lapins au n-ième mois

$F_0 = F_1 = 1$ Condition initiale

$$F_2 = 2$$

$$F_3 = 3$$

$$F_4 = 5$$

⋮

$$F_n = F_{n-1} + F_{n-2}$$

⋮

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,
10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040,
1346269, 2178309, 3524578, 5702887, 9227465, 14930352, 24157817, ...

[The On-Line Encyclopedia of Integer Sequences A000045](#)

LES LAPINS DE FIBONNACCI (SUITE)

Le calcul de F_n

FibOR(n, f_0, f_1)

Data: n entier, f_0, f_1 valeurs initiales

Result: F_n n -ième nombre de Fibonacci

if $n = 0$ **ou** $n = 1$ **then** Cas de base

if $n = 0$ **then**

return f_0

else cas $n = 1$

return f_1

else Récursion

return FibOR($n-1, f_0, f_1$)+FibOR($n-2, f_0, f_1$)

LES LAPINS DE FIBONNACCI (SUITE)

Le calcul de F_n

FiboR (n, f_0, f_1)

Data: n entier, f_0, f_1 valeurs initiales

Result: F_n n -ième nombre de Fibonacci

if $n = 0$ ou $n = 1$ then Cas de base

 if $n = 0$ then

 return f_0

 else cas $n = 1$

 return f_1

else Récursion

 return FiboR ($n-1, f_0, f_1$) + FiboR ($n-2, f_0, f_1$)

Complexité en nombre d'opérations + notée $C(n)$

$$C(n) = C(n-1) + C(n-2) + 1$$

$$C(0) = C(1) = 0$$

Ordre de grandeur ?

0,0,1,2,4,7,12,19,31,...

LES LAPINS DE FIBONNACCI (SUITE)

Ordre de grandeur

$$C(n) = C(n-1) + C(n-2) + 1; \quad C(0) = C(1) = 0$$

$$2C(n-2) \leq C(n) \leq 2C(n-1)$$

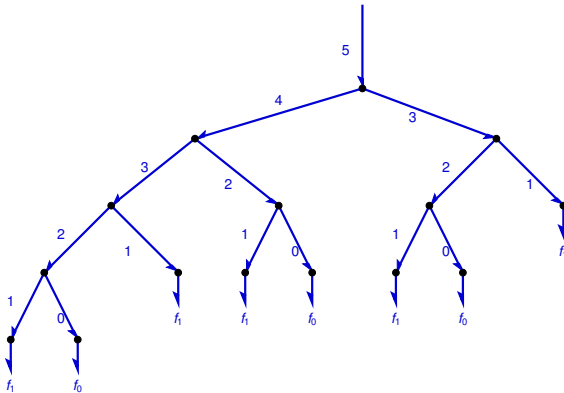
$$2^{n/2-1} C(2) \leq C(n) \leq 2^{n-1} C(2)$$

$$\log_2 C(n) = \Theta(n)$$

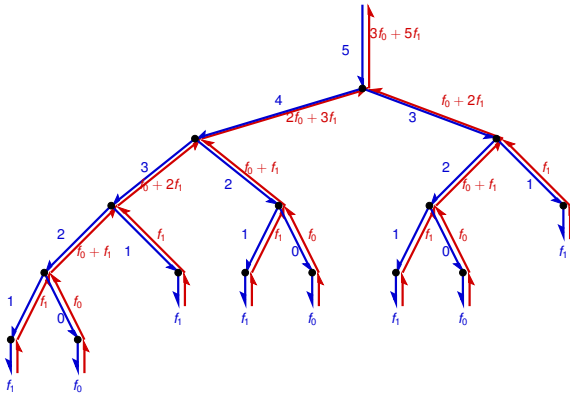
Peut-on faire mieux ?

Montrer que $C(n) = F_n - 1$

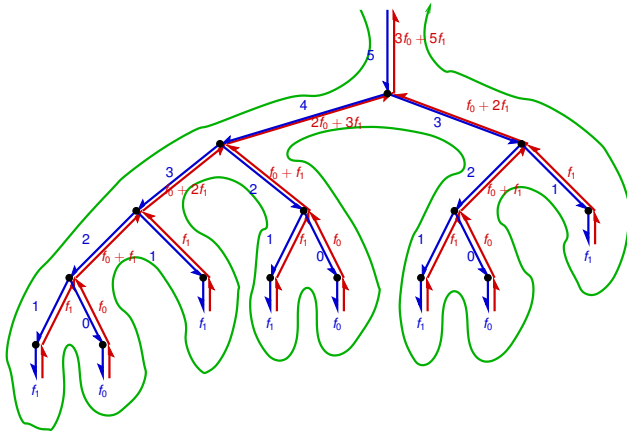
FONCTION RÉCURSIVE : ARBRE DES APPELS



FONCTION RÉCURSIVE : ARBRE DES APPELS

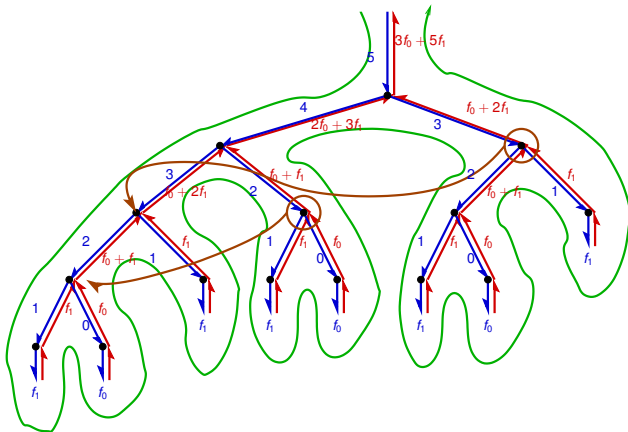


FONCTION RÉCURSIVE : ARBRE DES APPELS

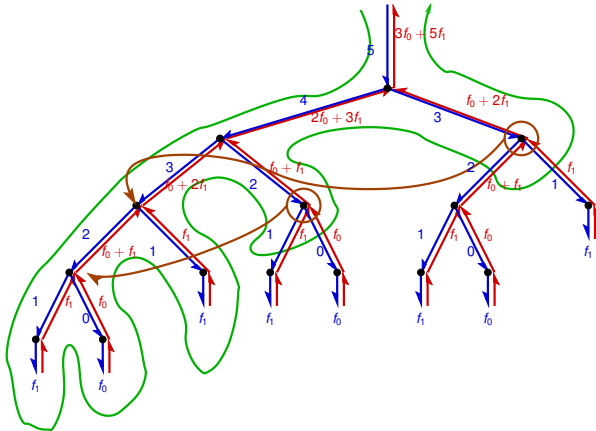


exécution = parcours de l'arbre en profondeur d'abord préfixé

FONCTION RÉCURSIVE : ARBRE DES APPELS



FONCTION RÉCURSIVE : ARBRE DES APPELS



memoisation = réutilisation des calculs intermédiaires

ÉCRITURE EN PYTHON

Écriture standard

```
def FiboR(n, f0=1, f1=1) :
    """
    Retourne le nieme nombre de Fibonacci
    parametres :
        f0, f1 : valeurs initiales,
        n : indice du nombre de Fibonacci
              a calculer,
        pre-condition : n entier, n>=0
        valeur de retour Fn
    Effet de bord : aucun
    Methode : appels recursifs
    >>> FiboR(3,1,1)
    3
    >>> FiboR(10,1,1)
    89
    """
    if (n == 0) : # cas de base
        return f0
    elif (n == 1) : # cas de base
        return f1
    else : # appels recursifs
        return FiboR(n-1, f0, f1) \
            + FiboR(n-2, f0, f1)
```

avec mémoïsation explicite

```
def fib(n, computed = {0: 1, 1: 1}):
    if n not in computed:
        computed[n] = fib(n-1, computed) \
            + fib(n-2, computed)
    return computed[n]
```

avec mémoïsation implicite

```
@functools.lru_cache(None)
def fib(n, f0=1, f1=1):
    if (n == 0) :
        return f0
    elif (n == 1) :
        return f1
    else :
        return fib(n-1, f0, f1) \
            + fib(n-2, f0, f1)
```

APPROCHE BOTTOM-UP

Formulation récursive de la résolution du problème

- ▶ explosion combinatoire possible (si facteur de division ≥ 2)
- ▶ redondance dans les appels récursifs (mémoïsation)
- ▶ structure dynamique de stockage
- ▶ réutilisation des résultats intermédiaires (cache de calculs)
- ▶ gestion de la mémoire (espace mémoire \longleftrightarrow temps de calcul)

Construction par extension

- ▶ Construction de la solution du problème à partir de la solution de tous les problèmes de taille inférieure (Bottom-up)
- ▶ $sol(n) = f(n, sol(0, \dots, n-1))$
- ▶ calculs inutiles, dépendant du schéma
- ▶ structure statique de stockage (tableau de solutions)
- ▶ **Programmation dynamique**
- ▶ Optimisation mémoire (durée de vie des solutions)

LES LAPINS DE FIBONNACCI (SUITE)

Le calcul de F_n

FiboI (n, f_0, f_1)

Data: n entier, f_0, f_1 valeurs initiales

Result: F_n n -ième nombre de Fibonacci

$F = f_0$

$G = f_1$

for $i = 2$ **to** n **do**

$H = F + G;$

$F = G$

$G = H$

return G

// valeur de F_i

LES LAPINS DE FIBONNACCI (SUITE)

Le calcul de F_n

FiboI (n, f_0, f_1)

Data: n entier, f_0, f_1 valeurs initiales

Result: F_n n -ième nombre de Fibonacci

$F = f_0$

$G = f_1$

for $i = 2$ **to** n **do**

$H = F + G;$

$F = G$

$G = H$

return G

// valeur de F_i

Complexité en nombre d'opérations + notée $C'(n)$

$$C'(n) = n - 1 = \Theta(n)$$

Peut-on faire mieux ?

ALGORITHMIQUE ET MODÉLISATION

- 1 DIVERTISSEMENT : les lapins
- 2 **CALCUL DE x^n : l'algorithme du débutant**
- 3 COMPLEXITÉ d'un problème

EXEMPLE : CALCUL DE $\underbrace{X * X * X * \dots * X}_{n \text{ FOIS}}$

Puissance (itérative)

PUISSANCE (x, n)

Data: Un réel x et un entier n

Result: Le calcul de x^n

```

p = 1
for i = 1 to n do
  p = p * x
return p
  
```

Coût en nombre d'opérations $*$ = n

Coût en espace mémoire = 4 (entiers)

Coût variable, dépend de la valeur des données

Taille des données en entrée : n (en fait $\log_2(n)$)

Peut-on faire mieux ?

Puissance (récursive)

PUISSANCE2 (x, n)

Data: Un réel x et un entier n

Result: Le calcul de x^n

```

if n = 0 then
  return 1
else
  return x * PUISSANCE2(x, n - 1)
  
```

EXEMPLE : CALCUL DE $\underbrace{x * x * x * \dots * x}_{n \text{ FOIS}}$ (2)

Un principe **Diviser pour régner**

PUISSANCE-DIV(x, n)

Data: Un objet x et un entier n positif

Result: La valeur de x^n

```

if  $n = 1$  then                                     // cas de base
  | return  $x$ 
else                                                // récursion
  | if  $n$  pair then
  | |  $z =$ PUISSANCE-DIV( $x, n/2$ )
  | | return  $z * z$ 
  | else
  | |  $z =$ PUISSANCE-DIV( $x, (n - 1)/2$ )
  | | return  $z * z * x$                                //  $n$  est impair  $\geq 3$ 

```

Coût de l'algorithme : $\mathcal{O}(\log n)$

Peut-on faire mieux ?

EXEMPLE : CALCUL DE $\underbrace{x * x * x * \dots * x}_{n \text{ FOIS}}$ (3)

Comme le nombre de façons de calculer x^n est fini. On peut construire l'*arbre des puissances*

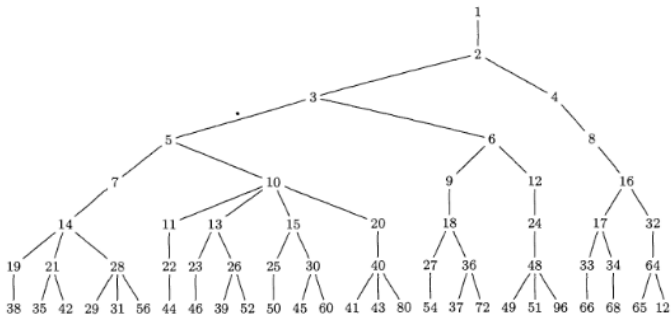


Fig. 14. The "power tree."

The Art of Computer Programming : Semi-Numerical Algorithms (vol 2) p464
 Pour calculer par exemple x^{29} on calcule $x^2, x^3, x^5, x^7, x^{14}, x^{28}$ et x^{29} .

EXEMPLE : CALCUL DE $\underbrace{x * x * x * \dots * x}_{n \text{ FOIS}}$ (4)

Un algorithme itératif (dérécursification)

The same idea applies, in general, to any value of n , in the following way: Write n in the binary number system (suppressing zeros at the left). Then replace each “1” by the pair of letters SX, replace each “0” by S, and cross off the “SX” that now appears at the left. The result is a rule for computing x^n , if “S” is interpreted as the operation of *squaring*, and if “X” is interpreted as the operation of *multiplying by x*. For example, if $n = 23$, its binary representation is 10111; so we form the sequence SX S SX SX SX and remove the leading SX to obtain the rule SSXSXSX. This rule states that we should “square, square, multiply by x , square, multiply by x , square, and multiply by x ”; in other words, we should successively compute $x^2, x^4, x^5, x^{10}, x^{11}, x^{22}, x^{23}$.

This binary method is easily justified by a consideration of the sequence of exponents in the calculation: If we reinterpret “S” as the operation of multiplying by 2 and “X” as the operation of adding 1, and if we start with 1 instead of x , the rule will lead to a computation of n because of the properties of the binary number system. The method is quite ancient; it appeared before 200 B.C. in Piṅgala’s Hindu classic *Chandaḥ-sūtra* [see B. Datta and A. N. Singh, *History of Hindu Mathematics* 2 (Lahore: Motilal Banarsi Das, 1935), 76]. There seem to be no other references to this method outside of India during the next 1000 years, but a clear discussion of how to compute 2^n efficiently for arbitrary n was given by al-Uqlīdisī of Damascus in A.D. 952; see *The Arithmetic of al-Uqlīdisī* by A. S. Saidan (Dordrecht: D. Reidel, 1975), 341–342, where the general ideas are illustrated for $n = 51$. See also al-Bīrūnī’s *Chronology of Ancient Nations*, edited and translated by E. Sachau (London: 1879), 132–136; this eleventh-century Arabic work had great influence.

The Art of Computer Programming : Semi-Numerical Algorithms (vol 2) p461

DONALD E. KNUTH



- ▶ The Art of Computer Programming (1962-73,2005)
- ▶ Combinatoire et Analyse d'algorithmes
- ▶ Literate Programming (1992)
- ▶ T_EX et Metafont
- ▶ Turing award 1974
- ▶ Langage Mix
- ▶ Algorithmes ...

Biographie de Donald E. Knuth *Mactutor History*

EXEMPLE : CALCUL DE $\underbrace{x * x * x * \dots * x}_{n \text{ FOIS}}$ (4)

Un principe **Décomposition du problème**

PUISSANCE-FACT(x, n)

Data: Un objet x et un entier n positif

Result: La valeur de x^n

```

if  $n = 1$  then                                     // cas de base
|   retourner  $x$ 
else                                               // récursion
|   if  $n$  est premier then
|   |    $z =$  PUISSANCE-FACT( $x, n - 1$ )
|   |   retourner  $x * z$ 
|   else                                           //  $n$  est composé  $n = p * q$ 
|   |    $y =$  PUISSANCE-FACT( $x, p$ )
|   |    $z =$  PUISSANCE-FACT( $y, q$ )
|   |   retourner  $z$ 

```

Coût de l'algorithme : ?

Peut-on faire mieux ?

EXEMPLE : CALCUL DE $\underbrace{x * x * x * \dots * x}_{n \text{ FOIS}}$ (5)

Théorème

La complexité du problème du calcul de la puissance est $\Theta(\log n)$.

- ▶ Il existe un algorithme de coût $\mathcal{O}(\log n)$ donc la complexité du problème est inférieure à $\mathcal{O}(\log n)$;
- ▶ D'autre part, montrons par récurrence la propriété
le coût minimal d'un algorithme de calcul de x^n est minoré par $\log_2 n$.

- Cette propriété est vraie pour $n = 1$ et $n = 2$.
- Supposons la propriété vraie pour tout $1 \leq i \leq n - 1$ et considérons l'algorithme optimal qui calcule x^n (il existe car il existe un nombre fini de manières de calculer x^n).

Cet algorithme effectue uniquement des opérations $*$.

La dernière exécution fait le produit $x^k * x^{n-k}$ pour une valeur de k que l'on peut choisir supérieure à $\frac{n}{2}$ (entre k et $n - k$ l'un des 2 est plus grand que $\frac{n}{2}$).

D'après l'hypothèse de récurrence le calcul de x^k demande au moins $\log_2 k$ opérations, c'est à dire au moins $\log_2 \frac{n}{2} = \log_2 n - 1$.

Donc le nombre d'opérations d'un algorithme optimal est minoré par $(\log_2 n - 1) + 1 = \log_2 n$.

Quelle moralité tirer du théorème ci-dessus ?

ALGORITHMIQUE ET MODÉLISATION

- 1 DIVERTISSEMENT : les lapins
- 2 CALCUL DE x^n : l'algorithme du débutant
- 3 **COMPLEXITÉ d'un problème**

COMPLEXITÉ D'UN PROBLÈME

Étant donné un problème \mathcal{P} , on appelle **complexité** du problème \mathcal{P}

$$C_{\mathcal{P}}(n) = \min_A C_A(n)$$

avec A algorithme qui résout \mathcal{P}

$C_A(n)$: la complexité au pire de l'algorithme A sur une instance de taille n

- ▶ La complexité d'un algorithme est un majorant de la complexité du problème qu'il résout
- ▶ plus difficile à calculer : ordres de grandeur, minorants, majorants
- ▶ le coût d'un algorithme particulier sur des données particulières ne donne aucune information sur la complexité du problème
- ▶ référence à une machine : coût des opérations

PROBLÈMES CLASSIQUES

- ▶ Recherche d'un élément dans un tableau de taille n (nombre de tests d'égalité) :
 $\mathcal{O}(n)$ parcours de l'ensemble
- ▶ Recherche d'un élément dans un tableau ordonné de taille n (nombre de comparaisons) :
 $\mathcal{O}(\log n)$ (recherche dichotomique)
- ▶ Calcul de x^n (nombre de multiplications) :
 $\mathcal{O}(\log n)$ (diviser pour régner)
- ▶ Tri d'un tableau (nombre de comparaisons) :
 $\mathcal{O}(n \log n)$ Heap-sort, Merge-sort...

COMPLEXITÉ DU PROBLÈME DU TRI

- ▶ **Problème** : Etant donné une suite d'éléments (x_1, x_2, \dots, x_n) dans un ensemble *ordonné* donner une permutation σ des indices telle que $(x_{\sigma(1)}, x_{\sigma(2)}, \dots, x_{\sigma(n)})$ soit ordonnée
- ▶ **Mesure de coût** : nombre de comparaisons
- ▶ **Borne supérieure** : Il existe des algorithmes de tri de complexité $\mathcal{O}(n \log n)$
- ▶ **Borne inférieure** : (hypothèse : les éléments sont différents)
 - La réponse est unique
 - Le nombre de permutations est $n!$
 - L'algorithme code une permutation
 - Il faut $\log_2 n!$ bits pour coder sans ambiguïté les $n!$ permutations
 - $\log_2 n! = \mathcal{O}(n \log n)$

ET LES LAPINS DE FIBONNACCI...

Quelle est la complexité du calcul de F_n ?

F_n = population de couples de lapins au n-ième mois

$F_0 = F_1 = 1$ Condition initiale

$F_n = F_{n-1} + F_{n-2}$

ET LES LAPINS DE FIBONNACCI...

Quelle est la complexité du calcul de F_n ?

F_n = population de couples de lapins au n-ième mois

$F_0 = F_1 = 1$ Condition initiale

$F_n = F_{n-1} + F_{n-2}$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} =$$

ET LES LAPINS DE FIBONNACCI...

Quelle est la complexité du calcul de F_n ?

F_n = population de couples de lapins au n-ième mois

$F_0 = F_1 = 1$ Condition initiale

$F_n = F_{n-1} + F_{n-2}$

$$\begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix}$$
$$=$$

ET LES LAPINS DE FIBONNACCI...

Quelle est la complexité du calcul de F_n ?

F_n = population de couples de lapins au n-ième mois

$F_0 = F_1 = 1$ Condition initiale

$F_n = F_{n-1} + F_{n-2}$

$$\begin{aligned} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-3} \end{pmatrix} \\ &\vdots \\ &= \end{aligned}$$

ET LES LAPINS DE FIBONNACCI...

Quelle est la complexité du calcul de F_n ?

F_n = population de couples de lapins au n-ième mois

$F_0 = F_1 = 1$ Condition initiale

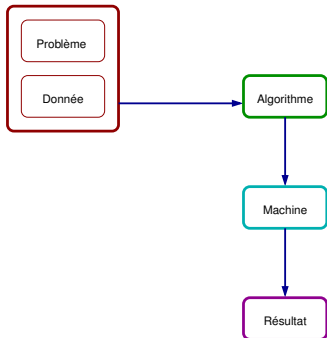
$F_n = F_{n-1} + F_{n-2}$

$$\begin{aligned} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{n-2} \\ F_{n-3} \end{pmatrix} \\ &\vdots \\ &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} \end{aligned}$$

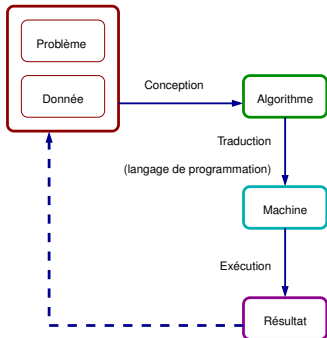
C'est un calcul de puissance

Youpi !

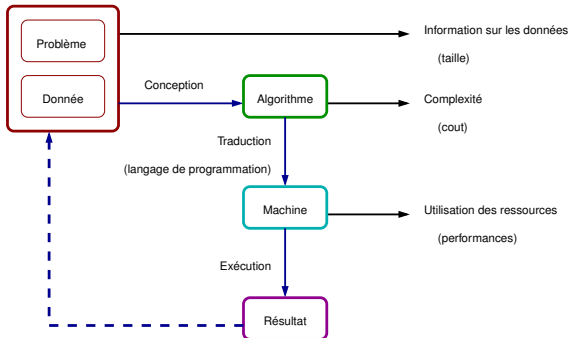
TAKE HOME MESSAGE



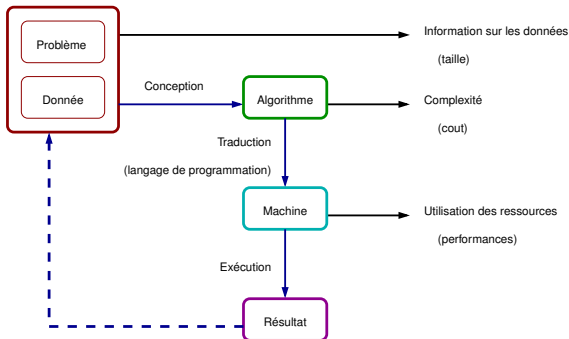
TAKE HOME MESSAGE



TAKE HOME MESSAGE



TAKE HOME MESSAGE



- ▶ Complexité de problème : forme (codage) des entrées / modèle de machine
- ▶ Complexité d'algorithme : ordre de grandeur en fonction d'un paramètre des données
- ▶ Coût d'un algorithme : comptage du nombre d'opérations

Modèle de production d'information en fonction des ressources utilisées