4 - Algorithmes de base

OBJECTIFS :

- Qu'est ce qu'un algorithme ?
- Définir une stratégie algorithmique
- démontrer la Terminaison
- Evaluer le Coût & Complexité

I - Qu'est ce qu'un algorithme ?

Du nom du mathématicen perse Al-Khawarizmi qui a inventé l'algèbre
[mots de même racine]

- Succession d'opérations à caractère répétitif/mécanique et qui aboutit à la résolution d'un problème :

ex : algorithme d'Euclide => trouver le PGCD

- Stratégie gagnante :

Pour résoudre un problème, gagner à un jeu

- Turing :

Tout algorithme est une machine de Turing

Un algorithme est «mécanisable»
donc il peut être traité par l'ordinateur.

Exemples simples :

- Recette de cuisine
- Partition de musique

Derrière tout algorithme, il y a un langage

- Les pas de danse :

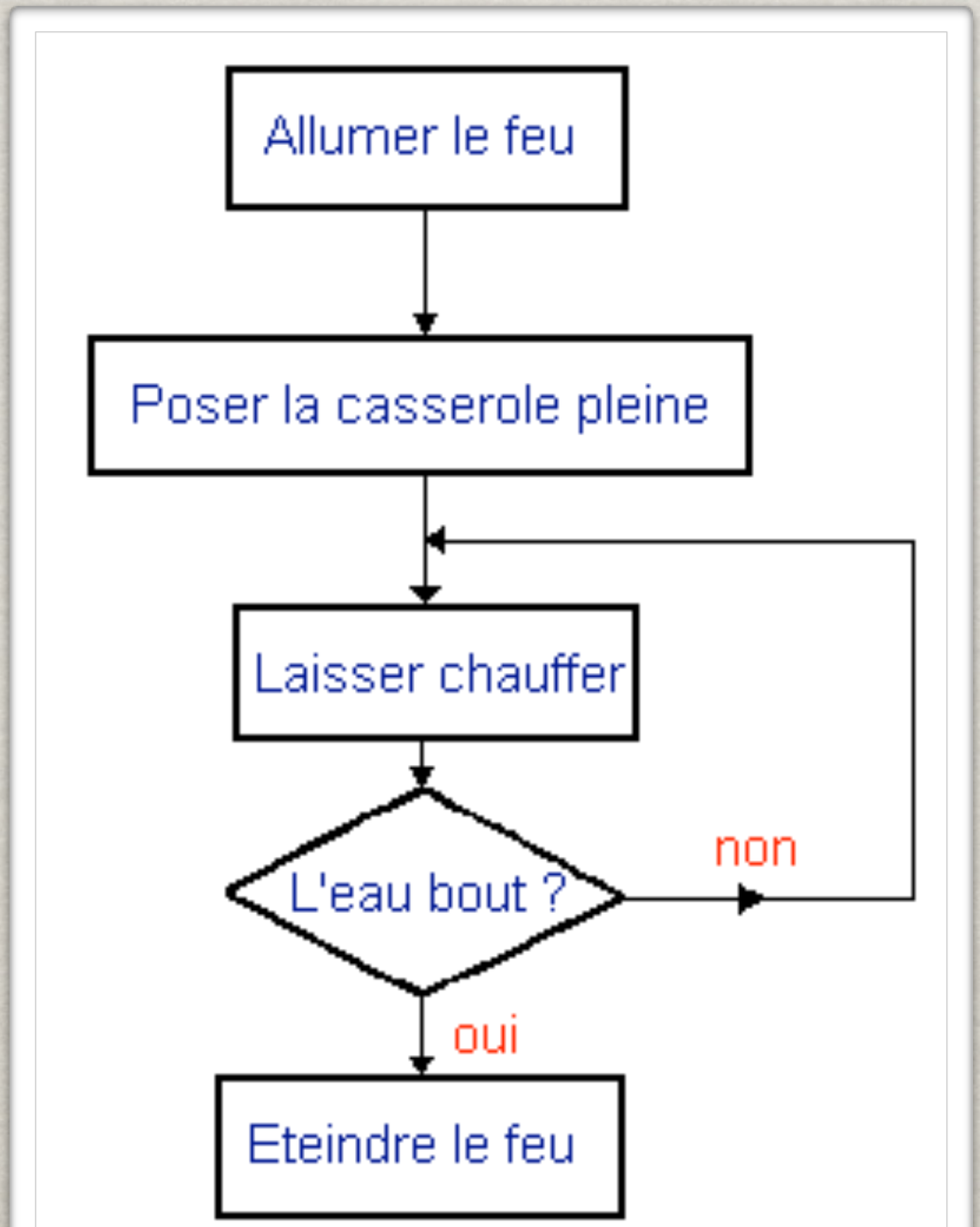
Le monde 2013 : Surveiller les algorithmes

Image : Dancing in the street par John Henderson.

- Horloge astronomique

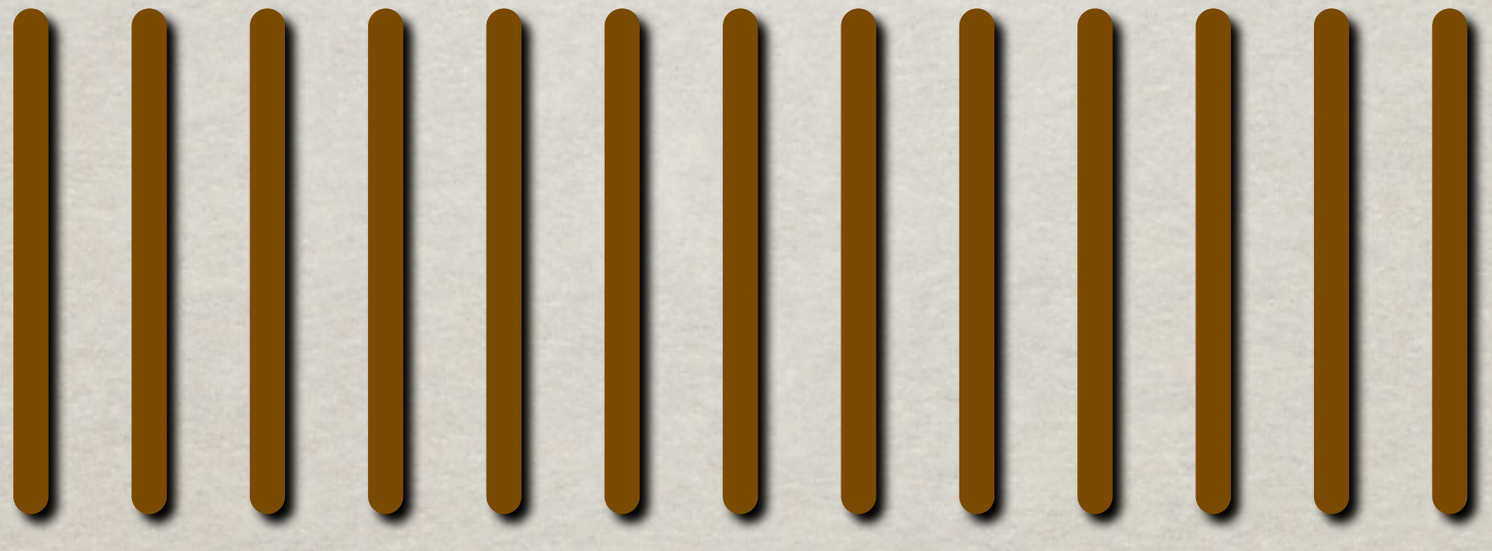
Succession de processus «mécaniques»

Ex : Algorithme pour faire bouillir de l'eau !!!

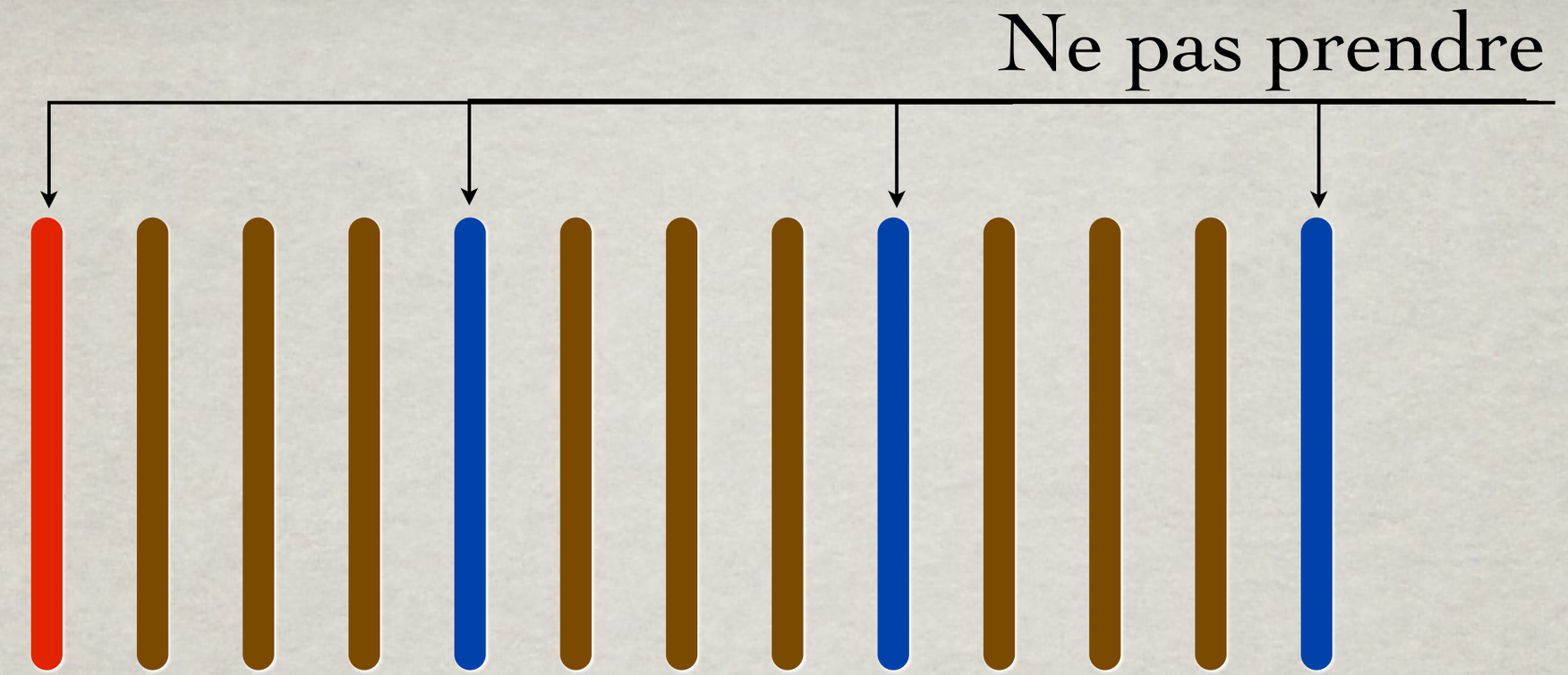


Application ludique :

Jeu des bâtons : chaque joueur prend à tour de rôle 1, 2 ou 3 bâton(s) [ni plus ni-moins]
il ne faut pas prendre le dernier bâton!



Exercice : Définir une stratégie gagnante.
Ecrire l'algorithme qui indique le nombre de bâton(s) a prendre à chaque coup.



Bâtons restant	1	2	3	4	5	6	7	8	9	10	11	12	13
reste / 4	1	2	3	0	1	2	3	0	1	2	3	0	1
prise	0	1	2	3	0	1	2	3	0	1	2	3	0

Sol° : Laisser 4n+1bâtons ==> prise de (N%4 - 1)%4 soit (N - 1)%4

<u>Premier algorithme :</u> Algorithme d'Euclide

Pour trouver le PGCD (plus grand commun diviseur de deux nombres A et B)

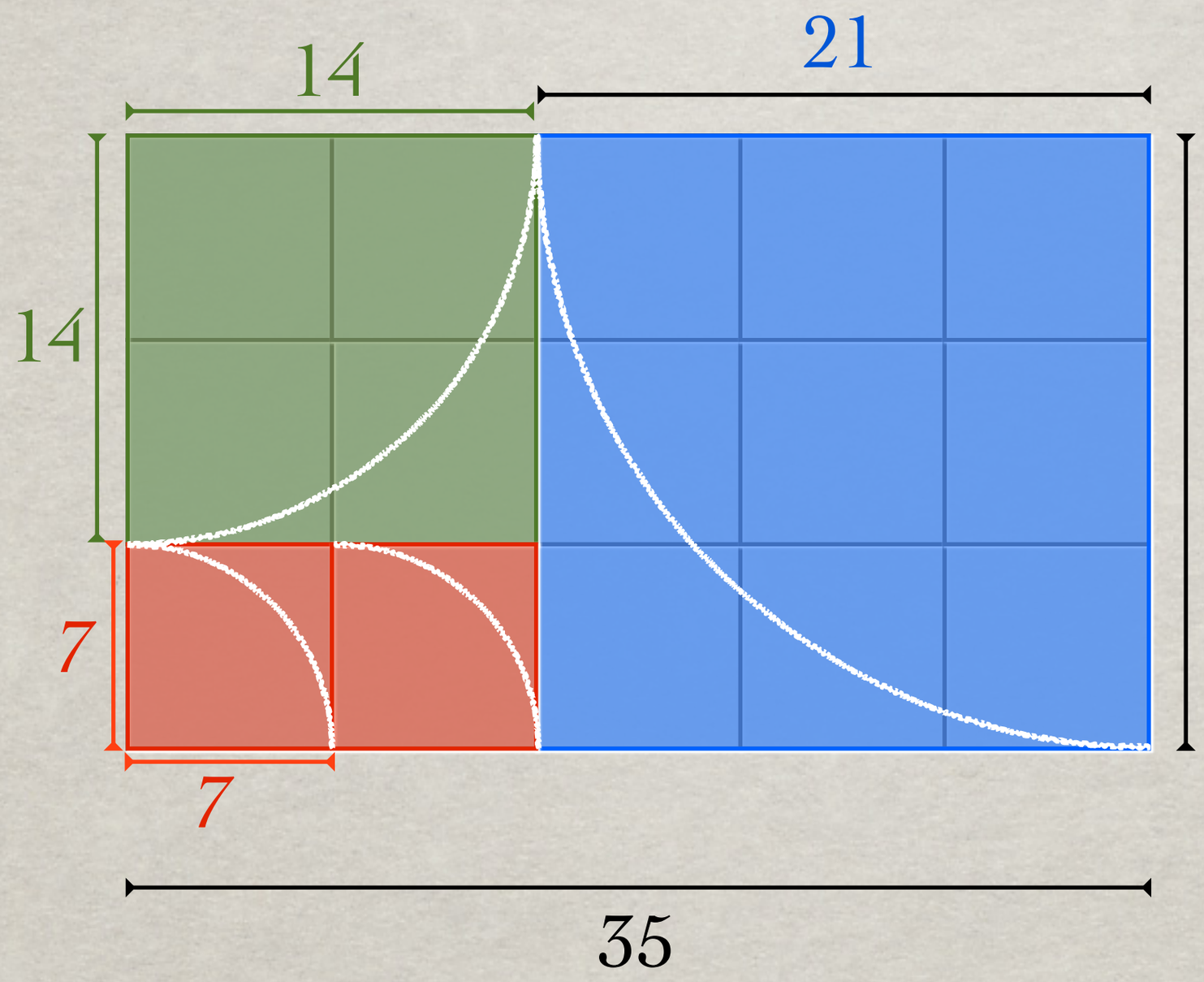
Soient A et B deux entiers naturels non nuls : $A \geq B$

On cherche le plus grand entier k qui divise A et B : $A = q \times k$

$B = q' \times k$

Comme k = 1 est solution triviale, k existe, mais quelle est la plus grande valeur possible pour k ? C-à-d son PGCD ?

Exemple : Algorithme d'Euclide pour (A,B)=(35, 21)



(35, 21)

(35, 21):14

(21, 14):7

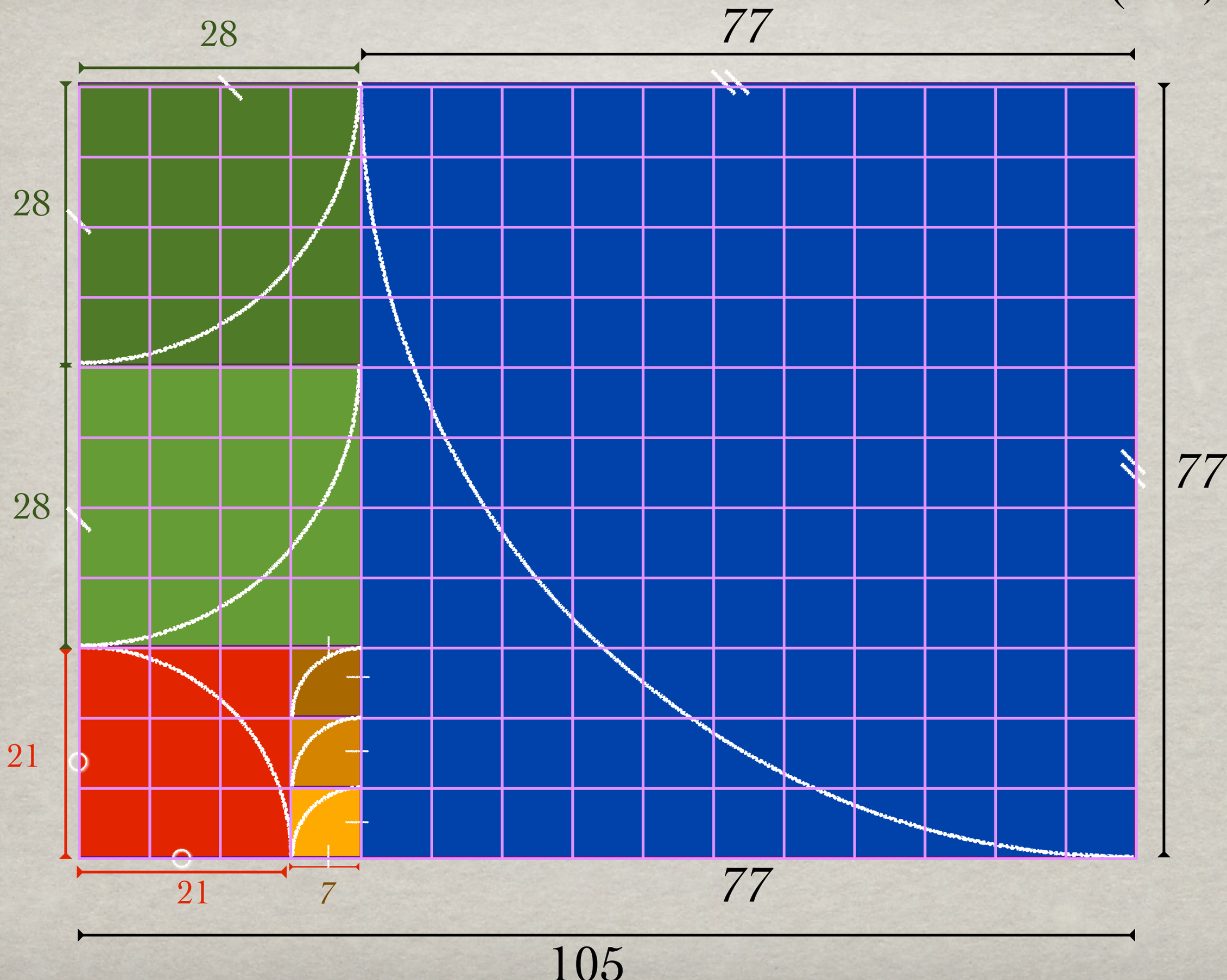
(14, 7):0

PGCD(35, 21) = 7

5x7 = 35

3x7 = 21

(105, 77)



(105, 77):28

(77, 28):21

(28, 21):7

(21, 7):0

PGCD(105, 77) = 7

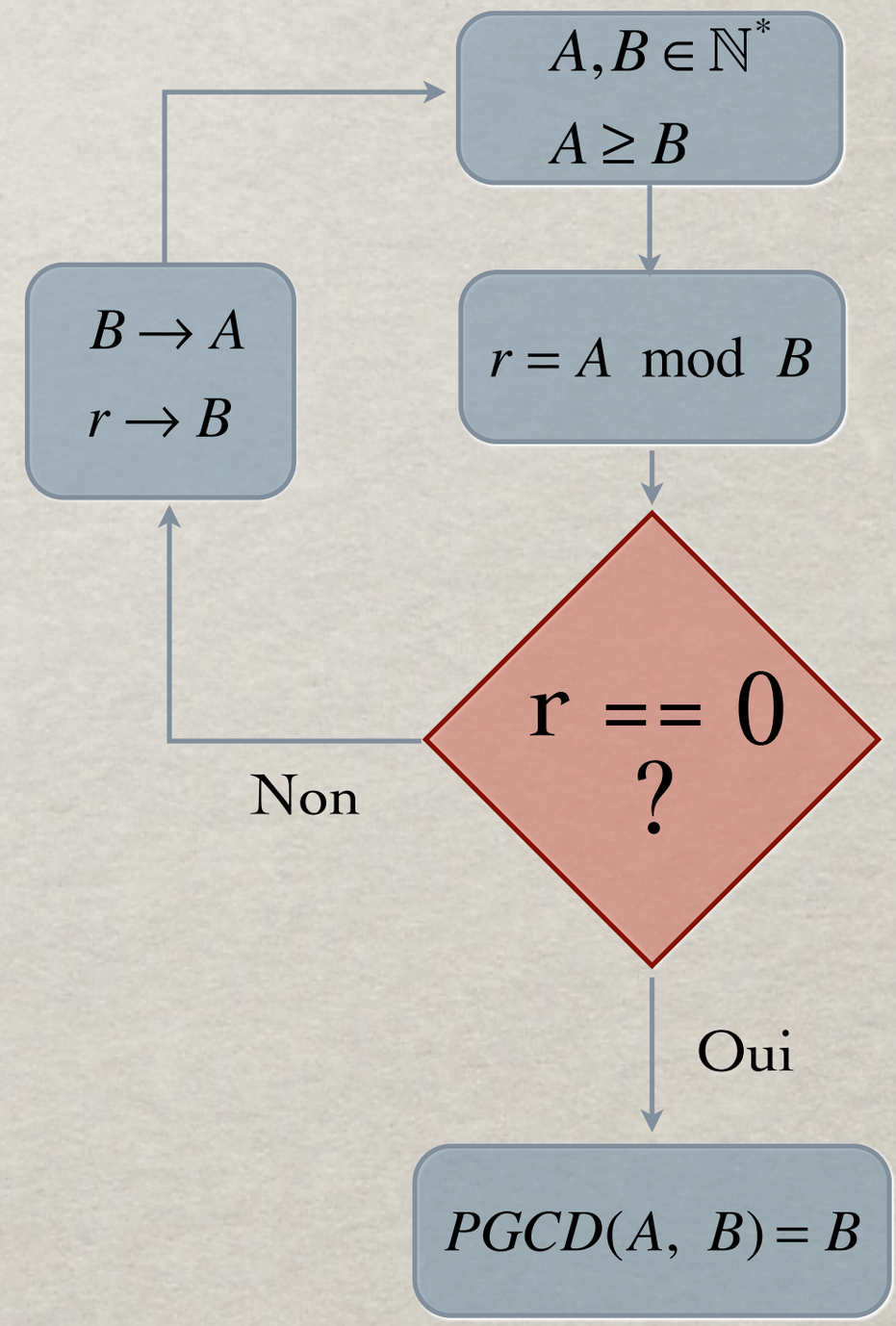
11x7 = 77

15x7 = 105

Schématisation de l'algorithme d'Euclide

Etape n :

$$A_n = q_n B_n + r_n$$



A chaque étape n, on note r_n le reste.
Si le reste est non nul :

B_n donne sa valeur pour A_{n+1}

r_n donne sa valeur pour B_{n+1}

Démonstration de l'algorithme d'Euclide

La démonstration s'appuie sur un **invariant de boucle**, c'est à dire une propriété qui reste vraie chaque itération et que nous allons démontrer : $PGCD(A,B) = PGCD(B,r)$

1 **Un peu d'arithmétique ….**

* Soit d un diviseur commun à A et B : d | A et d | B

$$\begin{cases} A = q_A d \\ B = q_B d \end{cases} \quad \text{or} \quad A = qB + r$$

alors $q_A d = q_B d \cdot q + r$ soit $r = d\left[q_A - q_B q\right]$ d'où d | r

Tout diviseur commun à A et B est diviseur du reste r de la division de A par B.
donc tout diviseur de A et B est diviseur de B et r, soit

$$\left\{d \mid d \mid A \text{ et } d \mid B\right\} \subset \left\{d \mid d \mid B \text{ et } d \mid r\right\}$$

* Soit d' un diviseur commun à B et r : d' | B et d' | r

$$\begin{cases} B = q_{B'} d' \\ r = q_r d' \end{cases} \quad \text{or} \quad A = qB + r$$

alors $A = q_{B'} d' \cdot q + q_r d' = \left[q_{B'} q + q_r \right] d'$ d'où d' | A

Tout diviseur commun à B et r est diviseur de A, donc tout diviseur de A et r est diviseur de A et B. soit :

$$\left\{ d \mid d \mid B \text{ et } d \mid r \right\} \subset \left\{ d \mid d \mid A \text{ et } d \mid B \right\}$$

Conclusion :

soient : $E_1 = \left\{ d \mid d \mid A \text{ et } d \mid B \right\}$ $E_2 = \left\{ d \mid d \mid B \text{ et } d \mid r \right\}$

Ces 2 ensembles sont égaux car contenus l'un dans l'autre et réciproquement :

$$E_1 \subset E_2 \text{ et } E_2 \subset E_1 \Rightarrow E_1 = E_2$$

2 Invariant de boucle :

$$PGCD(A,B) \equiv \sup\left(\left\{d \mid d \mid A \text{ et } d \mid B\right\}\right)$$

(par définition)

$$\sup\left(\left\{d \mid d \mid A \text{ et } d \mid B\right\}\right) = \sup\left(\left\{d \mid d \mid B \text{ et } d \mid r\right\}\right)$$

$$PGCD(A,B) = PGCD(B,r)$$

On posera donc :

(Tant que $r_n \neq 0$)

$$\begin{cases} A_{n+1} = B_n \\ B_{n+1} = r_n \end{cases} \quad \text{avec} \quad \begin{cases} A_0 = A \\ B_0 = B \end{cases}$$

L'algorithme prend fin dès lors que le reste devient nul, car on ne peut pas poursuivre : il faudrait envisager la division de A_n par 0.

On établit bien l'**invariant de boucle** suivant :

$$PGCD(A_n, B_n) = PGCD(A_{n+1}, B_{n+1})$$

Ainsi à chaque itération, le PGCD recherché est toujours le même ce qui **démontre l'algorithme**. Reste toutefois à prouver que la boucle **termine** pour obtenir le résultat.

Terminaison de l'algorithme d'Euclide

Notre approche est très générale, on démontre la terminaison en s'appuyant sur une **suite strictement décroissante et bornée par le bas**.

Soit la suite des restes dans la division euclidienne de A_n par B_n : $\boxed{(r_n) \equiv A_n \,//\, B_n}$ avec $r_0 = A \,//\, B$

<u>Pour tout n :</u> $r_n = A_n \,//\, B_n < B_n$ or $r_{n+1} < B_{n+1} = r_n$

Donc (r_n) est une suite strictement décroissante, et bornée par 0.

On a prouvé ainsi qu'il existe N tel que $r_N = 0$. $\boxed{PGCD(A,B) = B_N}$

<u>Remarque sur la **complexité en temps** :</u>

Combien d'itérations allons nous devoir faire pour arriver au résultat ?
On peut simplement ici garantir qu'il y aura moins de B itérations.

II - Les algorithmes de base

objectifs :

Dans toute étude d'un algorithme, on veut pouvoir :

1 - Etant donnée une stratégie algorithmique
=> «démontrer l'algorithme»

2 - démontrer la Terminaison

3 - Evaluer le Coût & Complexité
(temps et mémoire nécessaires)

1 - Définir une stratégie :

Il n'y a pas de règle générale, **la stratégie dépend de notre intuition**

[Il existe toutefois quantités de méthodes empiriques]

=> Souvent on commencera par des algorithmes dits «naïfs»

Il s'agit de passer en revue tous les cas possibles :
c'est un raisonnement par «induction»

=> Problème : ces algorithmes sont a priori peu performants

Pour le comprendre nous allons devoir quantifier la performance des algorithmes.

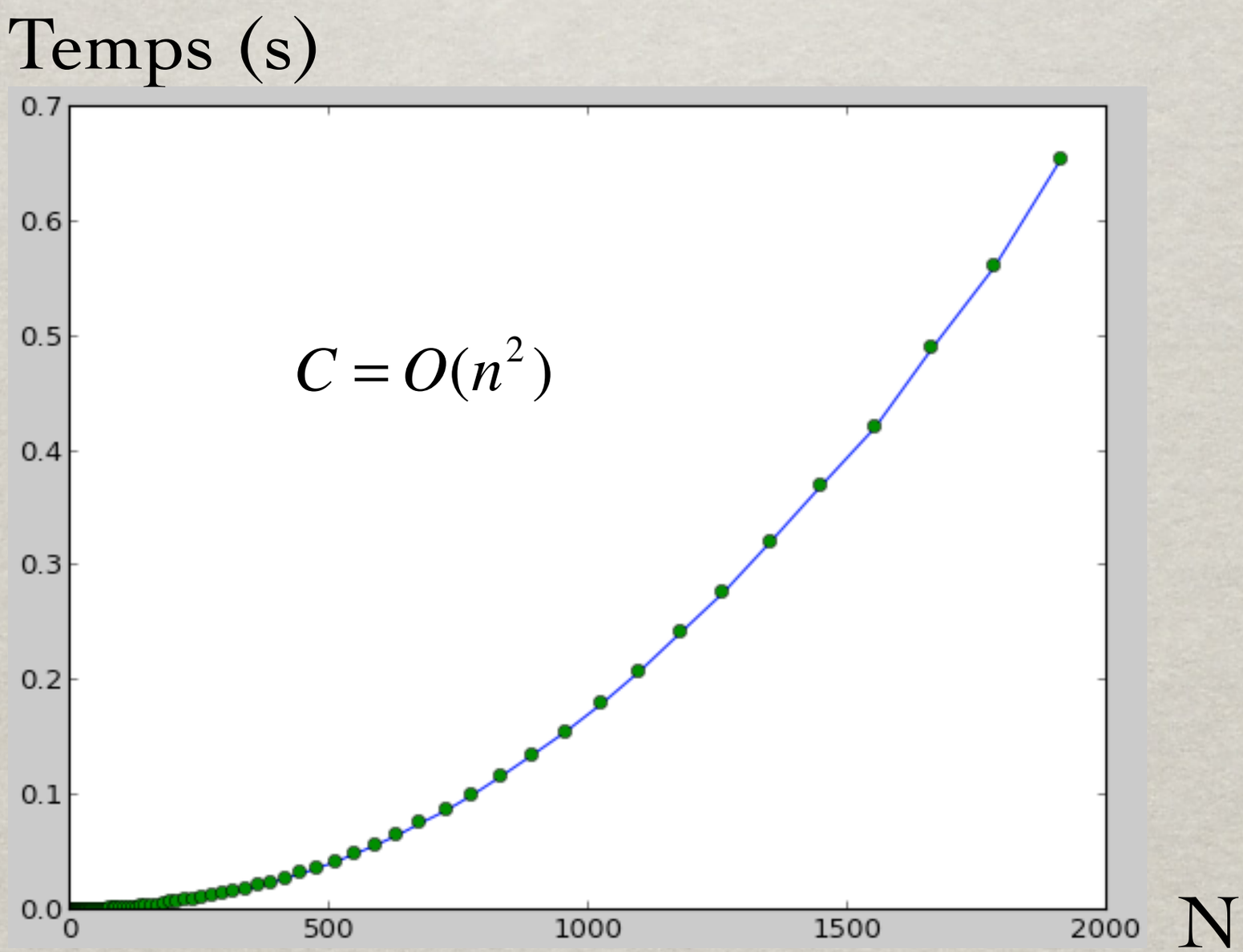
Deux critères de base peuvent intervenir, on parle de «coût» :

- L'algorithme «coûte» plus ou moins de mémoire. [stockage des données]
- L'algorithme «coûte» plus ou moins de temps.

Exemple : on a conçu un algorithme pour calculer tous les nombres premiers inférieurs à N

On constate que le temps nécessaire pour N=2000 est bien plus grand que celui pour N=1000
Comment cette tendance va évoluer si l'on cherche des nombres premiers très très grands ?

On peut mesurer le temps de calcul des nombres premiers inférieurs à N :



On observe que ce temps augmente en raison du carré de N !

Exercices :

exercice 1 - Soit un tableau d'entiers de taille N :

- Trouver un algorithme pour savoir si tous les entiers du tableau sont positifs.
- Dessiner le diagramme algorithmique.
- L'algorithme va t-il nécessairement avoir une fin ? on dit qu'il «termine».
- Combien d'opérations devez vous faire ?

exercice 2 - Algorithme «Naïf»

- Que fait l'algorithme suivant ? [commenter sa démarche]
- Dessiner le diagramme algorithmique.
- L'algorithme va t-il nécessairement avoir une fin ?
- Combien d'opérations réalise t-on ? (-> complexité)

```python
def algo(N):
    for i in range(2,N):
        PROP=True
        for j in range(2,i):
            if (i%j==0):
                PROP=False
        if (PROP):
            print("{:5d}".format(i), end='')

algo(1000)
```

```
  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67  71  73  79  83  89  97 101 103
107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233
239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379
383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523
541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677
683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853
857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991 997 ....
```

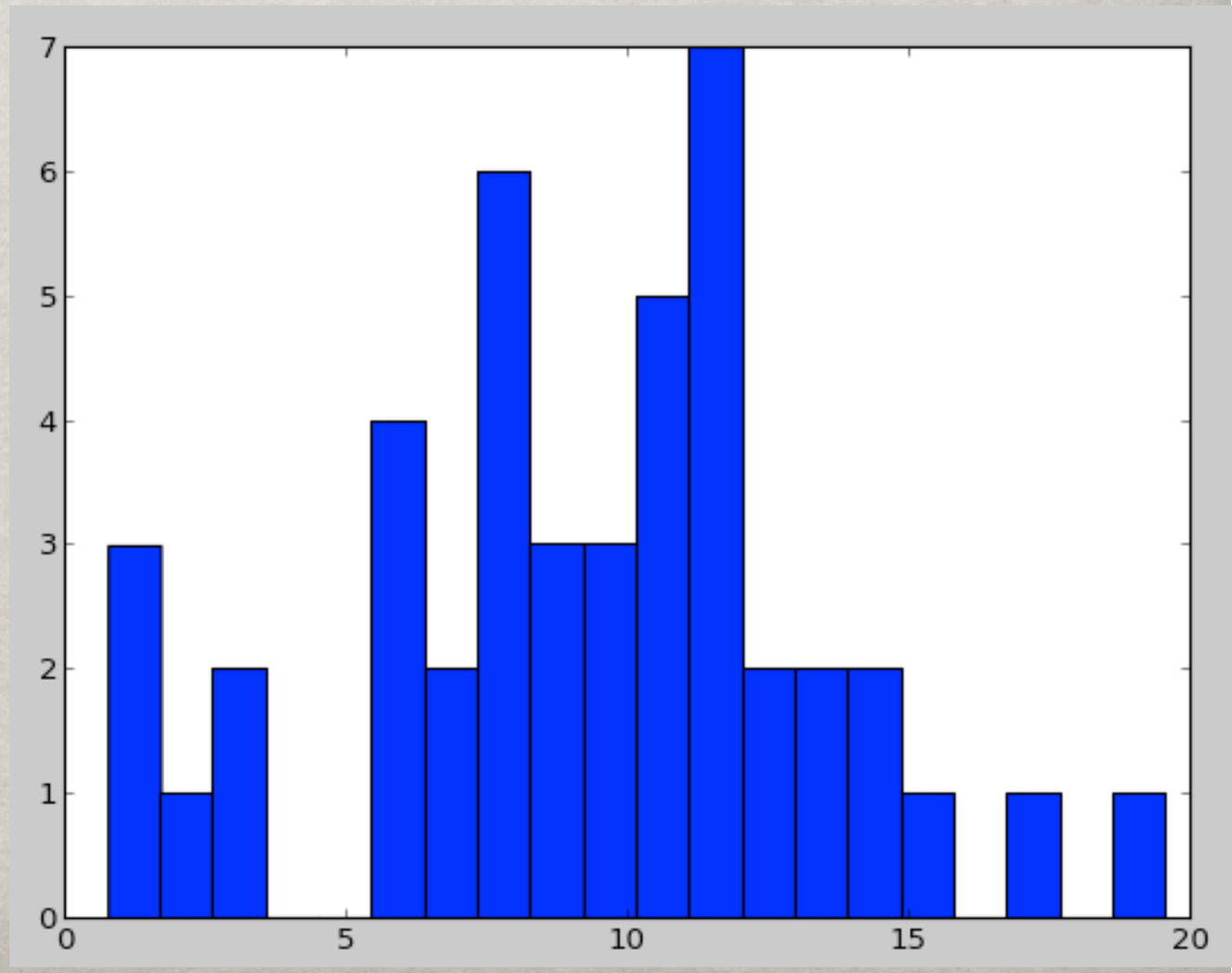
2 - Etude d'un tableau de réels

On étudie ici la liste des notes d'une classe :

- Les notes sont des réels entre 0 et 20 inclus.
- Il y a N élèves dans la classe

On peut générer des notes aléatoires en utilisant le module random de python. [help(random)]

```
import random as rnd

# Génération d'un tableau d'entiers aléatoires
# de taille N=45

note =[]

for i in range(N):
    note+=[round(rnd.gauss(8.0, 5.0), 3)]
print(note)

from matplotlib import pyplot as plt
plt.hist(note, bins=20)
plt.show()
```



notes :

note[30] ---> 19.565

Tableau de notes : note[i]

14.333,	13.51,	9.925,	6.558,	8.386,
11.412,	10.375,	6.196,	2.78,	6.615,
2.264,	14.609,	9.78,	17.207,	8.186,
7.951,	11.524,	11.795,	19.565,	7.922,
10.181,	12.301,	11.265,	7.422,	0.871,
10.264,	11.815,	11.11,	2.904,	15.188,
8.845,	10.745,	9.55,	12.37,	11.541,
7.894,	0.753,	5.742,	5.644,	3.033,
7.792,	1.016,	6.322,	10.548,	8.976,

Moyenne et écart-type

```
15 #Calcul de la moyenne
16 mu=0
17 for i in range(len(note)):
18     mu+=note[i]
19 mu=mu/N
20
21
22 #Calcul de la moyenne et de l'écart type
23 sig=0
24 for i in range(len(note)):
25     sig+=(note[i]-mu)**2
26 sig=(sig/N)**0.5
27
```

Exercice : le tri par sélection

Une idée simple permet de trier un tableau :

- On cherche le plus petit,
- puis le plus petit parmi ceux restant
- puis le plus petit parmi ceux restant
- etc …

- Proposer un algorigramme et un pseudo-code
- Rédiger le code Python
- Discuter la terminaison et le coût algorithmique

Algorithme de recherche de l'entier le plus grand : Max

Soit N la taille de mon tableau de note : «note».
Comment trouver la valeur la plus grande ?

On va passer en revue tous les éléments du tableau : méthode naïve

Algorithme «naïf» :

On initialise la valeur de Max :
--> premier élément

<u>Itération :</u>

On passe en revue tous les éléments, que l'on compare à la valeur de Max en cours.

On affiche la valeur de Max :

```
Max=note[0]

for i in range(1, len(note)):
    if note[i]>Max:
        Max=note[i]


print(Max)
```

N

Algorithme de recherche de l'entier le plus petit : Min

Proposer un ajustement très simple de notre algorithme pour trouver le minimum de notre tableau.

Exercice : Taper & Tester ce code chez vous

Démonstration et terminaison

On procède par récurrence, en s'appuyant toujours sur un invariant de boucle :

Invariant de boucle : Soit n l'étape en cours : $0 \leq n \leq N-1$

max est la valeur maximale du tableau de taille n+1 soit note[:n+1]

Initialisation : max = note[0] max est le seul élément => c'est bien le plus grand !

Itération : On suppose l'invariant vrai à l'étape n.
=> **max est la plus grande valeur de note[:n+1]**

Alors à l'étape n+1 :

note[n+1] > max => max = note[n+1]

note[n+1] ⩽ max => max inchangé !

Dans tous les cas, max est tjrs. la plus grande valeur de note[:n+2]

Ce qui valide l'invariant à l'étape n+1 !

Conclusion : **Par récurrence, la propriété est vraie ∀ n**, et **après N-1 itérations l'algorithme Termine** la boucle for, ce qui démontre l'algorithme.

max sera bien la plus grande valeur du tableau !

Rq : la boucle **for** garantit que l'algorithme termine : le nombre d'itérations étant connu dès le départ.

Comment mesurer le coût en temps ?

1 Opération élémentaire →

Boucle sur note : **(N-1) fois**

2 Opérations élémentaires
(même temps pour chaque itération)

- 1 Comparaison
- 1 affectation

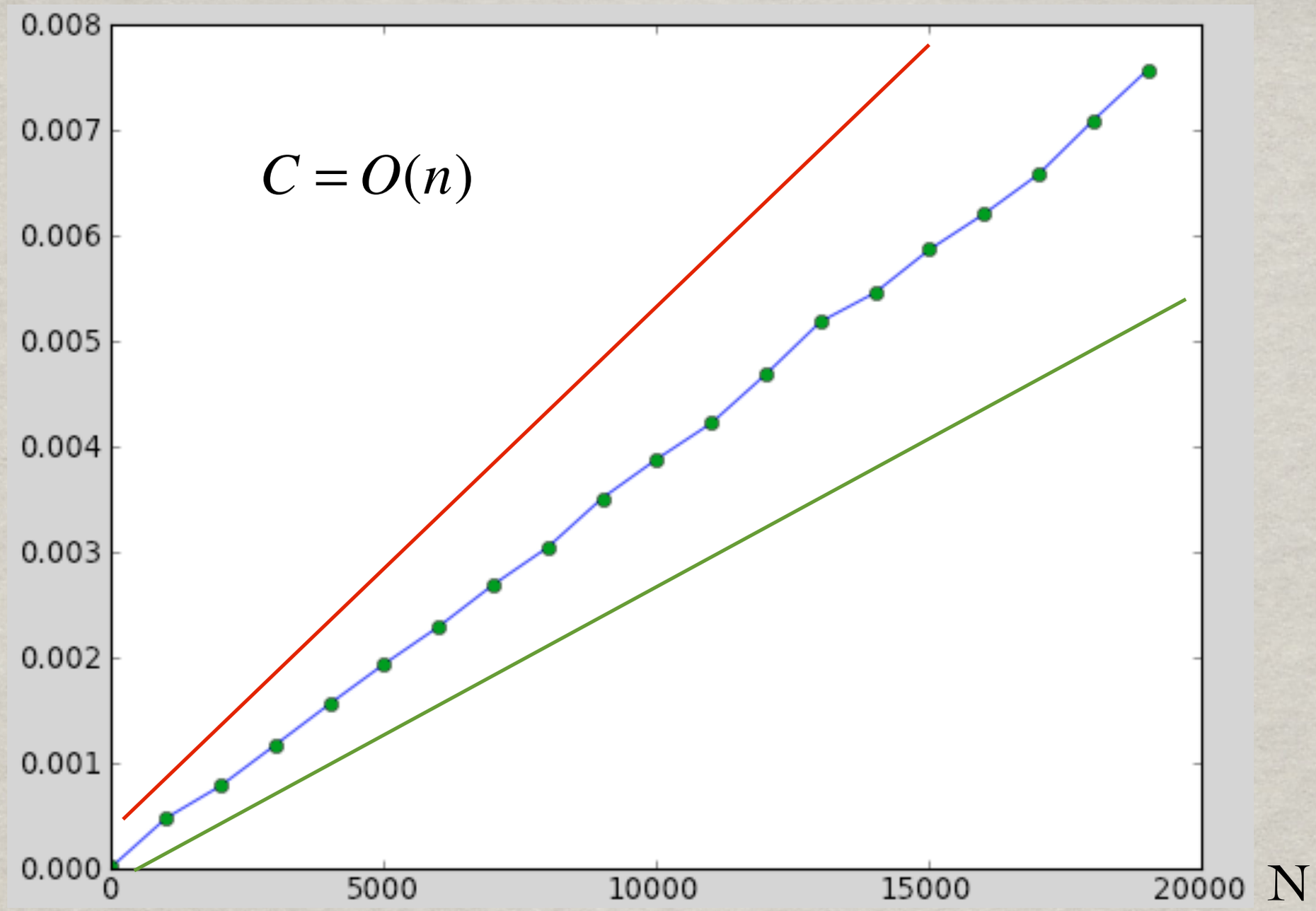
```
Max=note[0]

for i in range(1, len(note)):

    if note[i]>Max:
        Max=note[i]

print(Max)
```

N

1 Opération élémentaire →

<u>Calcul de la compléxité :</u> $\mathbb{C} =$

Temps (s)



Complexité linéaire :

On peut toujours trouver des bornes sup et inf linéaires, c-à-d qu'il existe :

- une droite (en rouge) supérieure pour tout N à notre temps de calcul.
- une droite (en vert) inférieure pour tout N à notre temps de calcul.

Exercice : produit matriciel

On représente une matrice par une liste de lignes, chaque ligne contenant les valeurs de ses colonnes :

```
      [ [1, 2, 3],                    [ [2, 9, 4],
A =     [4, 5, 6],              B =     [7, 5, 3],
        [7, 8, 9] ]                     [6, 1, 8] ]
```

Rq syntaxe : A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

Proposez un code simple qui réalise le produit matriciel de deux matrices A et B quelconques et vérifier sur le cas particulier ci-dessus.

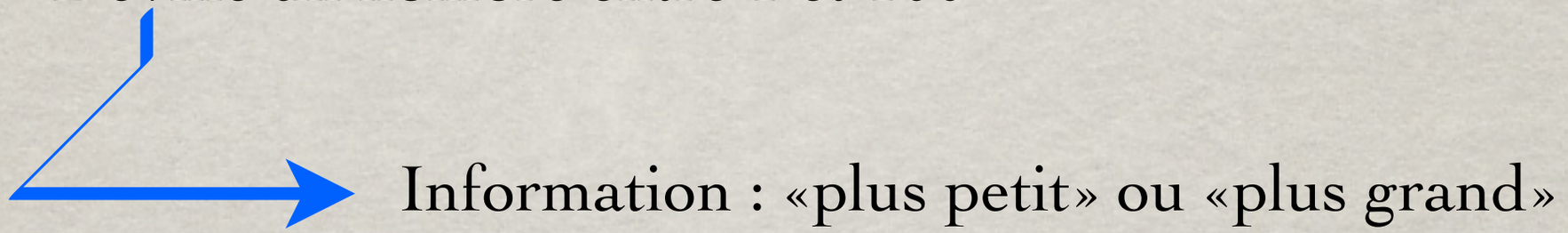
Écrire une fonction produit(A, B) qui prend en argument les listes de listes telles que ci-dessus et qui renvoie la matrice produit sous la même forme. On peut faire l'hypothèse que les matrices sont de taille 3 puis généraliser.

3 - Recherche dans un tableau trié

Ce point est essentiel et trouve beaucoup d'applications, en particulier pour la gestion de base de données.

L'objectif est ici de déterminer où se trouve un élément du tableau sur lequel on a une information simple

Exemple : «Devine un nombre entre 1 et 100»



Exercice : Trouver deux algorithmes possibles

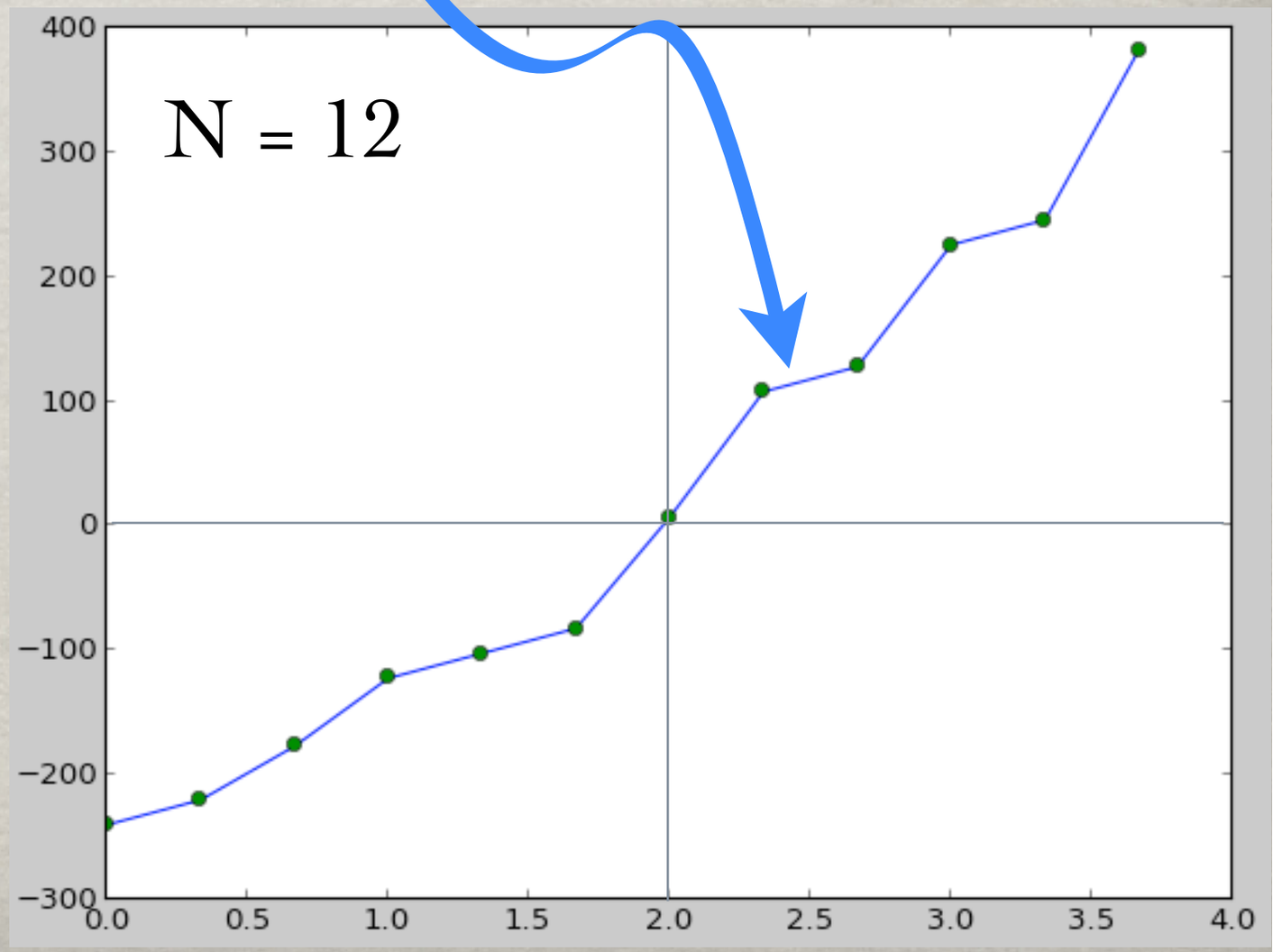
Algorithme : recherche d'un zéro par dichotomie

Informations :
- la fonction est croissante
- «positif» ou «négatif»
  [c-à-d premier bit 0 ou 1 => opération élémentaire très rapide]

On génère au préalable une **fonction croissante aléatoire**
dans un tableau de taille N :

On génère une fonction croissante,
à l'aide d'une marche aléatoire,
dans un tableau de taille N

La fonction est créée pour que l'on passe une seule fois du négatif au positif et jamais l'inverse



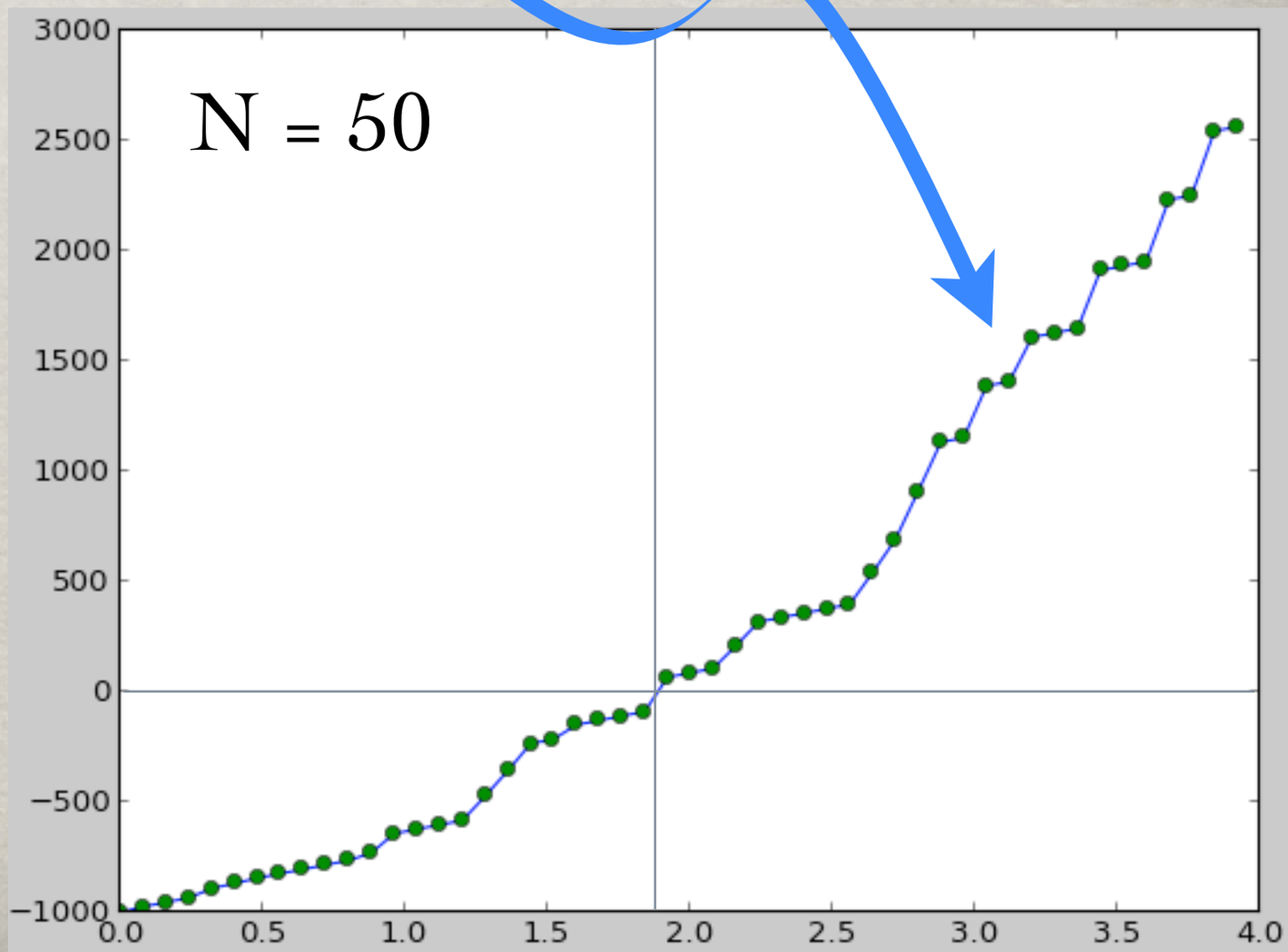
Algorithme : recherche d'un zéro par dichotomie

Informations :
- la fonction est croissante
- «positif» ou «négatif»
  [c-à-d premier bit 0 ou 1 => opération élémentaire très rapide]

On génère au préalable une **fonction croissante aléatoire** dans un tableau de taille N :

On génère une fonction croissante, à l'aide d'une marche aléatoire, dans un tableau de taille N

La fonction est créée pour que l'on passe une seule fois du négatif au positif et jamais l'inverse



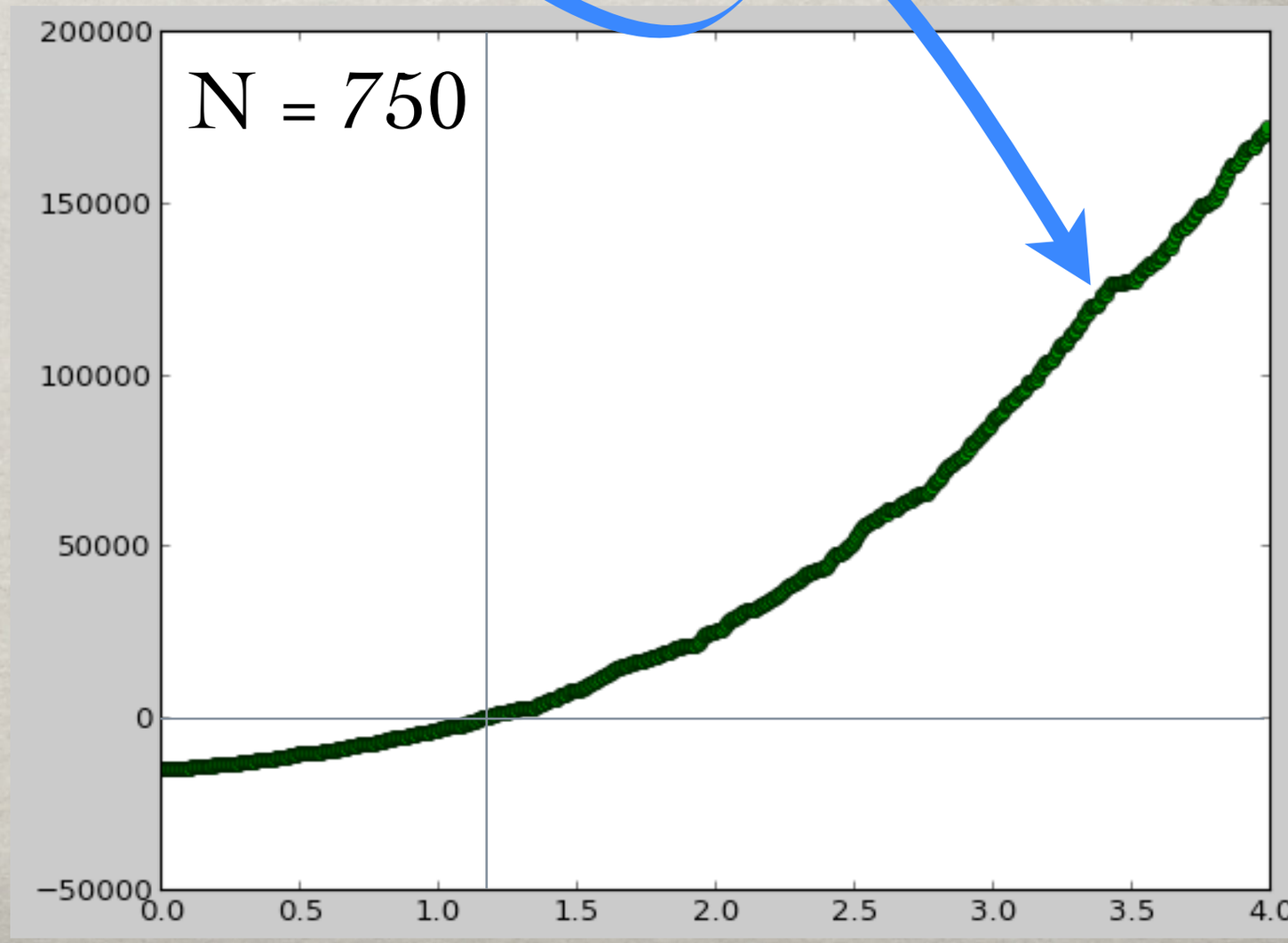
Algorithme : recherche d'un zéro par dichotomie

Informations :
- la fonction est croissante
- «positif» ou «négatif»
  [c-à-d premier bit 0 ou 1 => opération élémentaire très rapide]

On génère au préalable une **fonction croissante aléatoire** dans un tableau de taille N :

On génère une fonction croissante, à l'aide d'une marche aléatoire, dans un tableau de taille N

La fonction est créée pour que l'on passe une seule fois du négatif au positif et jamais l'inverse



Soit T un tableau de taille N : aucun élément du tableau ne vaut exactement zéro !

=> On recherche les indices g et d respectivement à gauche et à droite du zéro :
C-à-d tels que $T(g) < 0$ et $T(d) > 0$ car la fonction est croissante

=> la fonction nous renvoie également l'intervalle des x où la fonction s'annulle.
On en déduira un encadrement sur x de la position du zéro !

Création du tableau aléatoire

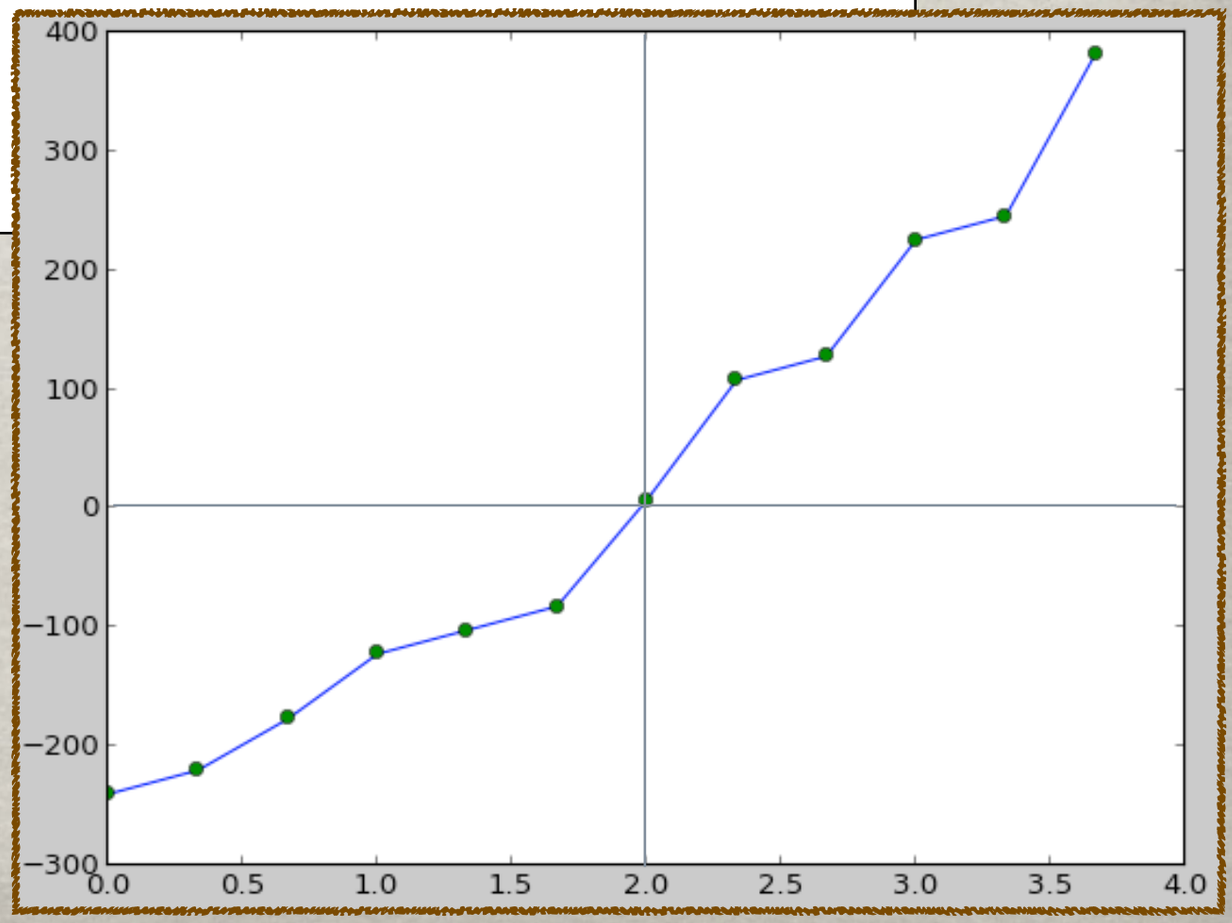
```
import random as rnd

def  TAB(N):
    xmin=0.0;       xmax=4.0

       tabx=[xmin];    taby=[-20*N]

    for i in range(1,N):
       x = xmin + i/N*(xmax - xmin)   ##Abscisse : extrapolation linéaire entre xmin et xmax
       tabx+=[x]
       taby+=[ taby[-1] - taby[0]/N      ##Marche aléatoire (très arbitraire...)
                + 2/(N**0.5)*(rnd.randrange(2)-1)*taby[0]*(N/i)**(rnd.randrange(2)-2)]

  return (tabx, taby)
```

TAB(N) renvoie donc un tuple dont les deux éléments sont des tableaux de taille N :

- le premier élément donne : les abscisses x -> tabx
- le second élément donne : les ordonnées y -> taby

Utilisation du module matplotlib.pyplot

```
#Tracé de la fonction
from matplotlib import pyplot as plt

N=75
Tplot = TAB(N)  ## Chaque appel de TAB(N) régénère une fonction aléatoire
                ##  => il faut ne l'appeler qu'une fois et affecter le résultat !

Tx=Tplot[0]     ##  On isole le tableau des abscisses et celui des ordonnées.
Ty=Tplot[1]
plt.plot(Tx,Ty)
plt.plot(Tx,Ty,'o')
plt.show()
```



Algorithmes Naïf de recherche du zéro de la fonction

Boucle sur Tab : **N' - fois**

1 Opérations élémentaires
(même temps pour chaque itération)

3 Opérations élémentaires
[mais une seule fois !]

1 Comparaison

2 affectations

1 stop !

2 Opérations élémentaires

```
#recherche du zéro par la méthode naïve
def zero(tab):

    for i in range(len(tab[0])-1):

        if (tab[1][i]*tab[1][i+1]<=0):
                g=i
                d=i+1
                break

    #Affichage des indices et abscisses
    print(g,tab[1][g])
    print(d,tab[1][d]);
```

N

Démonstration et terminaison

Terminaison : On a une boucle for => Nombre d'itération connu : ça termine !

Complexité : $\mathbb{C} = N'(b_1 + b_2 + b_3) + a_1 + a_2 < a + bN$ $\boxed{\mathbb{C} = O(N)}$

Démonstration : Invariant trivial : **« Je n'ai pas encore trouvé le zéro »**
—> vrai initialement : avant la boucle.

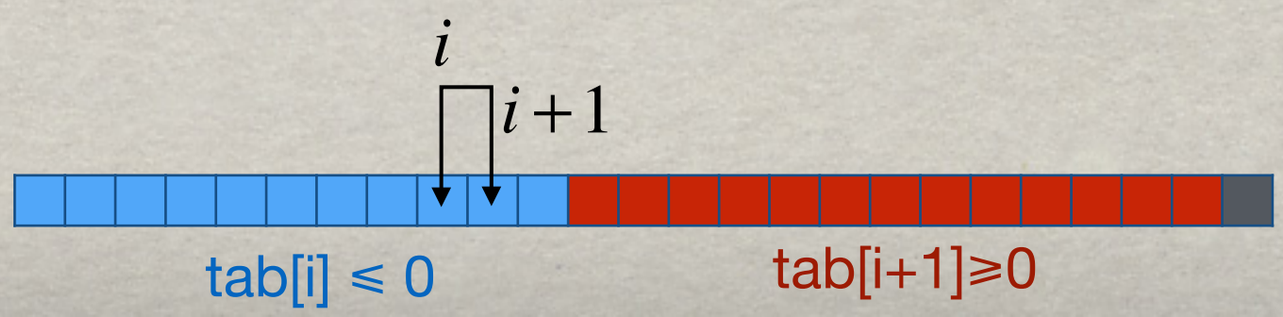
Par construction, on sait que les valeurs sont strictement croissantes :
=> il existe donc une position i telle que tab[i] ⩽ 0 et tab[i+1]⩾0.
=> cette position est unique.

Soit une itération i :

$i \in [\![0, N - 2]\!]$

si tab[i] * tab[i+1] > 0 => pas de zéro invariant vrai pour i+1

si tab[i] * tab[i+1] ⩽ 0 => Zéro trouvé : STOP !



Algorithmes de recherche du zéro par dichotomie

2 Opérations élémentaires

Boucle sur Tab : **? - fois**

3 Opérations élémentaires
(même temps pour chaque itération)

2 Opérations élémentaires

```
#recherche du zéro par dichotomie
def dicho(tab):

    g=0;  d=len(tab[0])-1

1 Comparaison  while (g<d-1):
1 Comparaison      if (tab[1][(g+d)//2]>0):
                       d=(g+d)//2
1 affectation      else:
                       g=(g+d)//2

    #Affichage des indices et abscisses
    print(g,tab[1][g])
    print(d,tab[1][d])
```

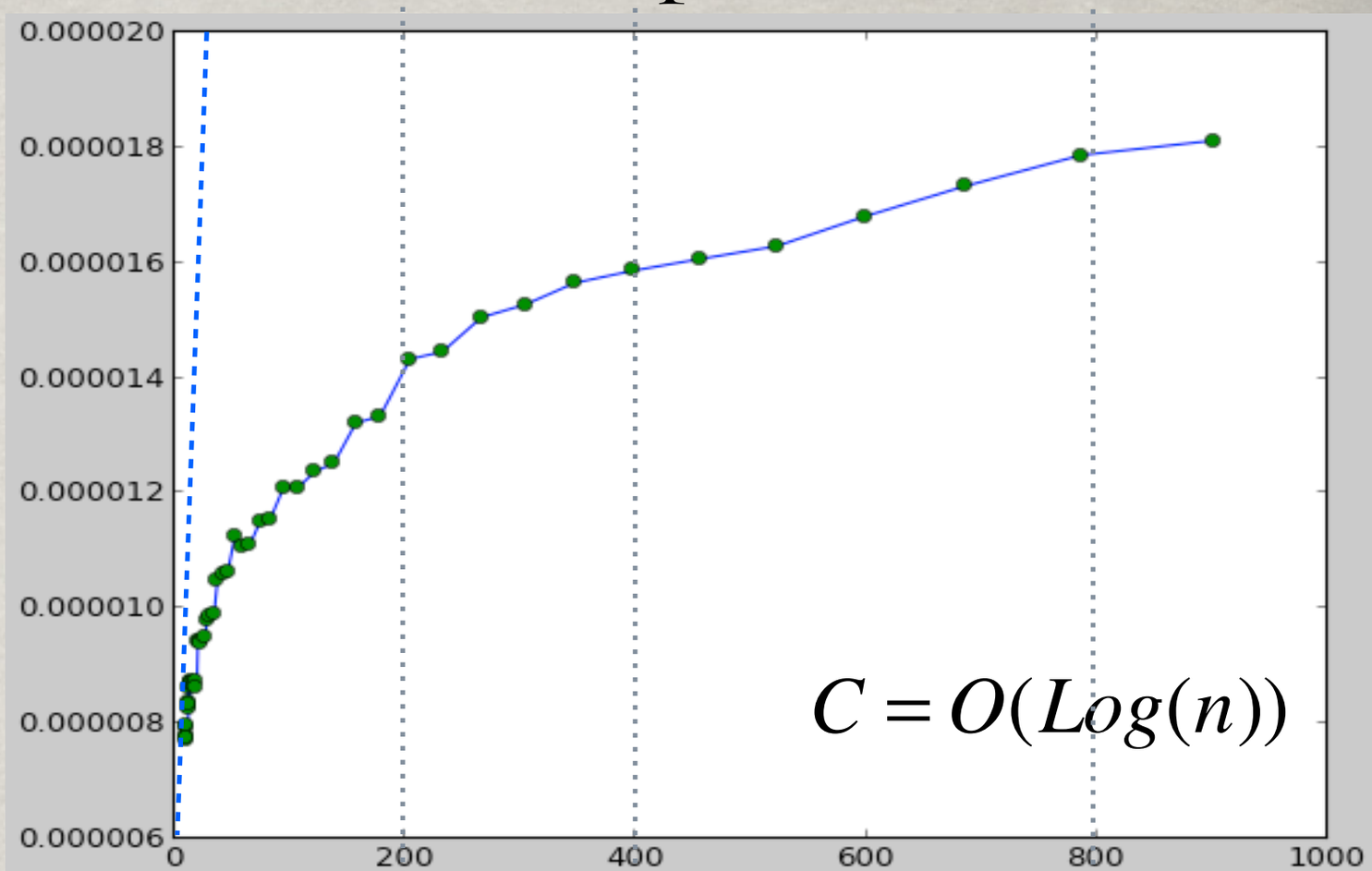
N

La complexité réside donc dans le nombre d'itérations !

Comment l'évaluer ?

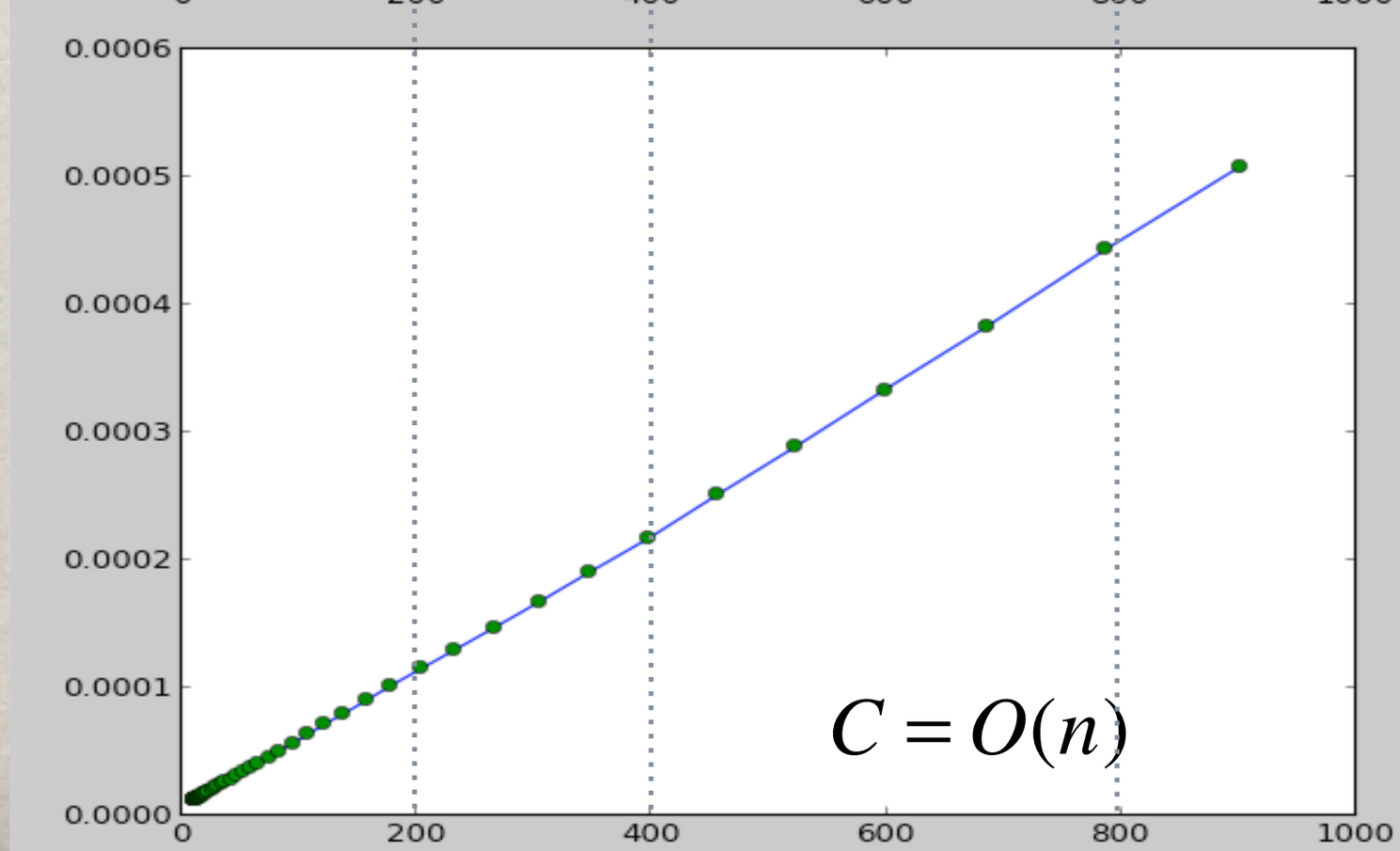
Chaque point est une expérience aléatoire répétée 1000 fois !

Dichotomie



Naïf

N = 200 : 7 fois plus long
N = 400 : 12 fois plus long
N = 800 : 26 fois plus long



Conclusions :

- Pour des petits tableaux, les temps sont très courts et la méthode naïve convient très bien.

- Plus la taille du tableau augmente plus la méthode par dichotomie devient interessante.

- Pour de très grands volumes de données la méthode par dichotomie devient indispensable pour réaliser une recherche dans un temps acceptable :

N	ZERO	DICHO
10^5	0.0096	0.0001309
10^6	0.055	0.00017
10^7	0.32	0.000369
10^8	BUG !	

Démonstration et terminaison

Terminaison : Soit Tn la suite des tailles du tableau : T0 = N

$$\forall n > 2 \quad T_{n+1} \leq T_n / 2 + 1 < T_n$$

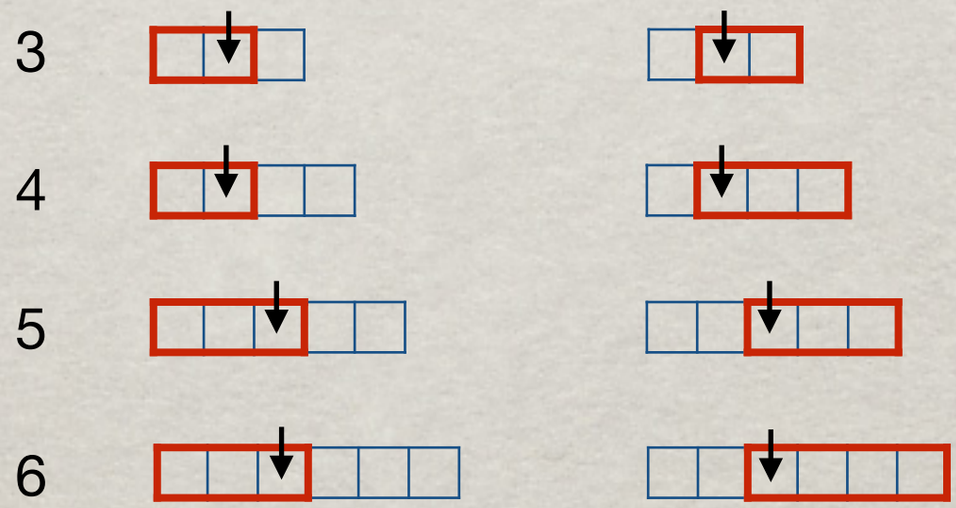
Si n ⩽ 2 on a trouvé le zéro.

Attention :

On note : **/**
La division euclidienne

La suite Tn est ainsi strictement décroissante et bornée par le bas en 2 :
donc **l'algorithme termine !**

Combien d'étapes :



3 —> 2
4 —> 2
4 —> 3 —> 2
5 —> 3 —> 2
6 —> 3 —> 2
6 —> 4 —> 2
6 —> 4 —> 3 —> 2

16 —> 9 —> 5 —> 3 —> 2
16 —> 8 —> 5 —> 3 —> 2
16 —> 8 —> 4 —> 3 —> 2
Séquence la plus courte : 16 —> 8 —> 4 —> 2

Conclusion :

Au final, et même pour de très grands nombres N le nombre total d'itérations k ne pourra changer que d'une unité !

$$\text{int}(Log_2(N)) - 1 \leq k \leq \text{int}(Log_2(N)) + 1$$

Approximation des grands nombres :

La complexité n'a pas de valeur à l'unité près.
Soient N >> 1 la taille du tableau et k >> 1 le nombre d'étapes.

On a typiquement : $T_{n+1} = T_n / 2$ et à la fin : $T_k = 2$

$T_{k-1} = 2 \times 2$ $T_{k-2} = 2 \times 2 \times 2$ $T_0 = T_{k-k} = 2 \times 2 \times \cdots \times 2 = 2^{k+1}$

Soit : $N = 2^{k+1}$ $k+1 = Log_2(N)$ $\boxed{k = Log_2(N) - 1}$

Rq : Ce calcul correspond au meilleur des cas,
On sait qu'en réalité il peut y avoir une étape de plus voire 2.

Dans l'approximation des grands nombres, on retiendra que : $\boxed{k \simeq Log_2(N)}$

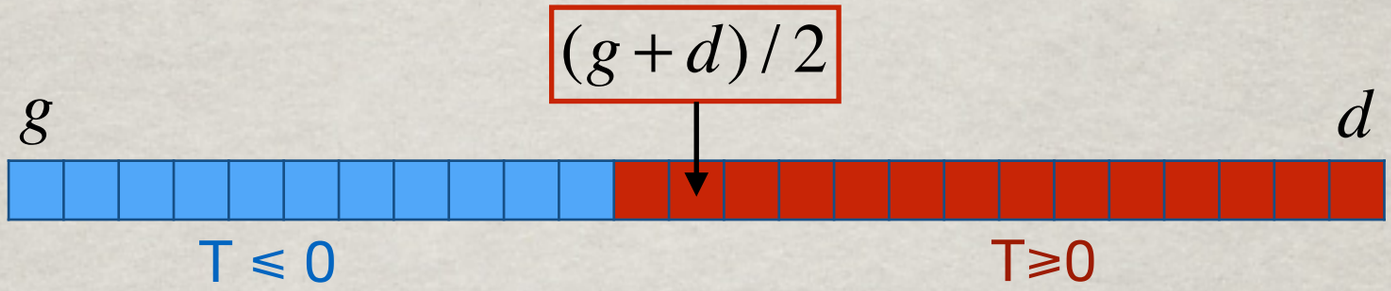
Complexité : $\mathbb{C} = a_1 + a_2 + k(b_1 + (b_2) + (b_3)) + a_3 + a_4$

$\mathbb{C} \leq a + bk \leq a + bLog_2(N)$

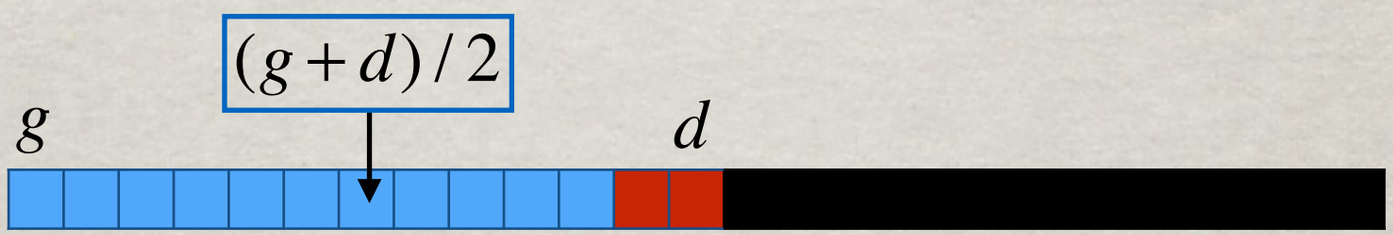
Soit : $\boxed{\mathbb{C} = O(Log_2(N))}$ C'est la complexité dans le pire et le meilleur des cas.

Exemple de déroulement :

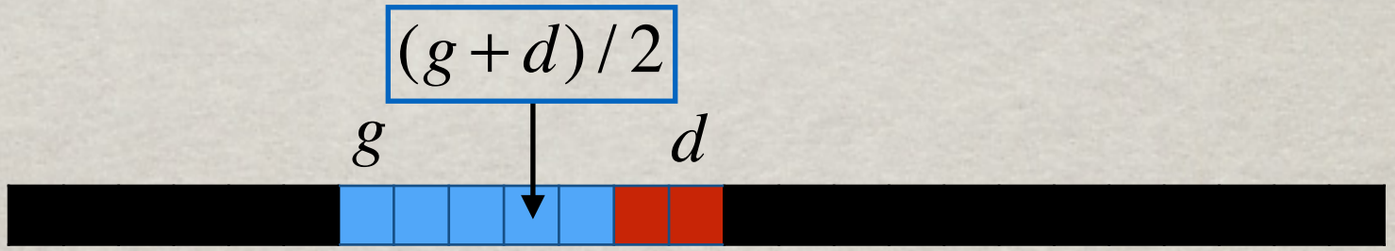
$T_0 = 25$



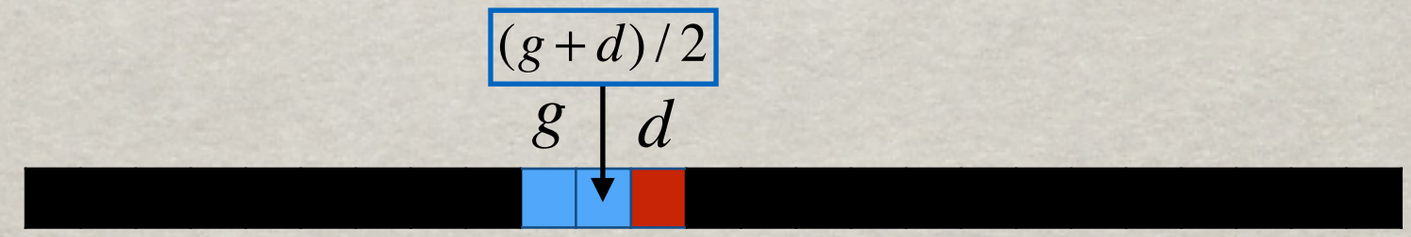
$T_1 = 13$



$T_2 = 7$



$T_3 = 3$

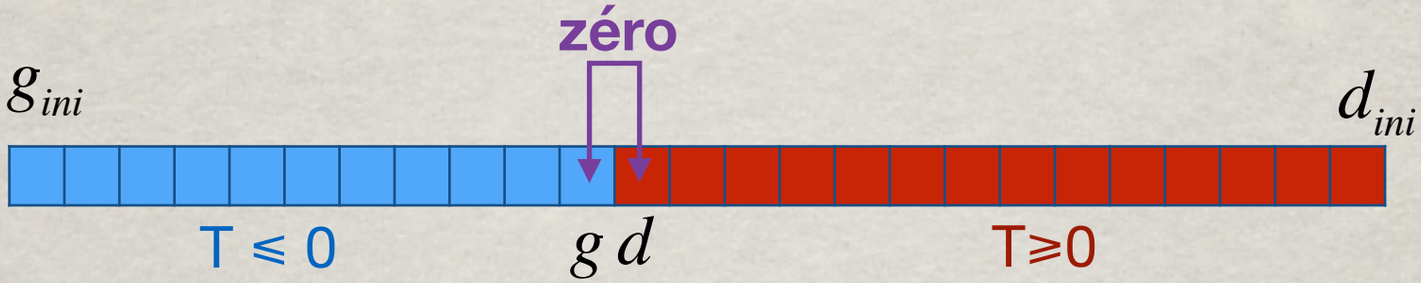


$T_4 = 2$

STOP

Démonstration :

On note T le tableau des valeurs (=> donc les ordonnées de Tab)
Le zéro est la donnée de g et d : indices encadrant le changement de signe, tels que d = g+1.



On pose l'**invariant de boucle** suivant :

- Le zéro est dans **T[g : d+1]**
- **g < d** (boucle ssi g < d -1)

Initialisation : **Le zéro est dans T[0 : N] par hypothèse (avec N >1)**
0 < N-1

Check !

Propagation : Invariant vérifié à l'étape n et **g < d-1** car on rentre ans la boucle.

Soit **T[(g+d)/2] > 0** => On déplace la frontière droite : $d_{new} = (g+d)/2$ $g_{new} = g$

Check !

- Le zéro est dans $T[g_{new} : d_{new} + 1]$
- $g = \frac{g+g}{2} < \frac{g+d-1}{2} \leq (g+d)/2 = d_{new}$ $g_{new} < d_{new}$

Soit **T[(g+d)/2] ⩽ 0** => On déplace la frontière gauche : $g_{new} = (g+d)/2$ $d_{new} = d$

Check !

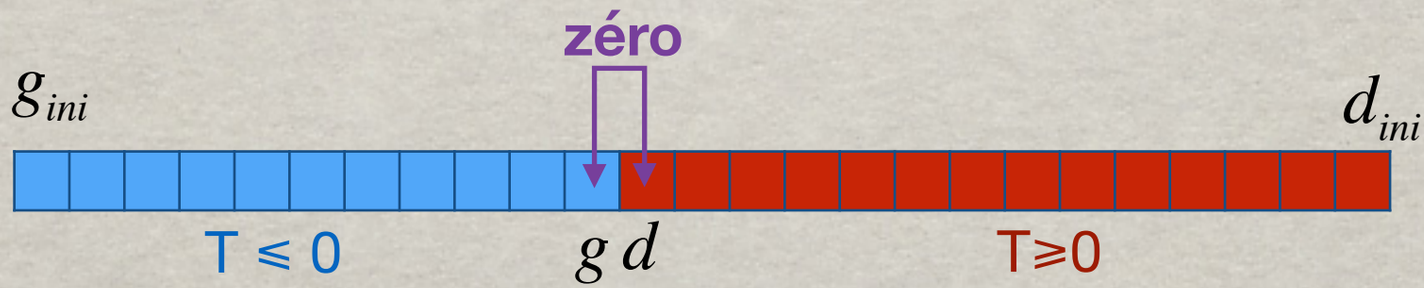
- Le zéro est dans $T[g_{new} : d_{new} + 1]$
- $g_{new} \leq \frac{g+d}{2} < \frac{d-1+d}{2} = \frac{2d-1}{2} < d = d_{new}$ $g_{new} < d_{new}$

Dans tous les cas l'invariant est vérifié à l'étape n + 1 :

- si il était vérifié à l'étape n
- et qu'on est entré dans la boucle. (g < d - 1)

Condition d'arrêt : **Le programme s'arrête si l'on ne peut plus entrer dans la boucle.**
soit **g ⩾ d - 1**. Or l'invariant nous dit aussi que **g < d**.

Conclusion : **g = d - 1** autrement dit : g et d encadrent le changement de signe !



On a démontré l'algorithme !

Exercices : Diviser pour régner !

Tous en cuisine !

On doit découper une tranche de jambon que l'on supposera rectangulaire en petits carrés ou rectangles : comment faire ?

- proposer une méthode naïve
- une méthode diviser pour régner

montrer que la méthode diviser est bien plus intéressante et ce d'autant plus que le nombre de petits carrés est important ! Miam Miam …

Divide & Conquer => I Got the Power !

On souhaite calculer la puissance n d'un nombre a : $\boxed{a^n}$

n est quelconque, on ne connait pas sa parité, et il faut réaliser n-1 multiplications.
- Proposer un algorithme qui subdivise le problème en deux pour limiter le nombre d'opérations
- A l'aide d'une fonction récursive (c-à-d qui se rappelle elle-même) réaliser un algorithme optimisant le calcul de a puissance n.
- Quelle est la complexité de ce calcul ? (ODG du nombre de multiplications réalisées.)

Exercices : Diviser pour régner !

L'âge du Capitaine 2

Nous avons résolu le problème de l'âge du capitaine avec un algorithme naïf dans un temps en O(N^2).

En utilisant la recherche dichotomique proposer un algorithme qui sera plus rapide :

- On écrira une fonction ageCapitaine qui renvoie tous les couples solutions.
- On écrira une fonction dichoFind qui dit si elle trouve une valeur complémentaire. Cette seconde fonction est utilisée par la première

```
def ageCapitaine(tabValeurs, valeurCible):
    ########
    return sol

def dichoFind(tabValeursTriee, valeurCible):
    ########
    return vrai ou faux
```