4 - Algorithmes de base

1 - Définir une stratégie :

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

=> 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» :

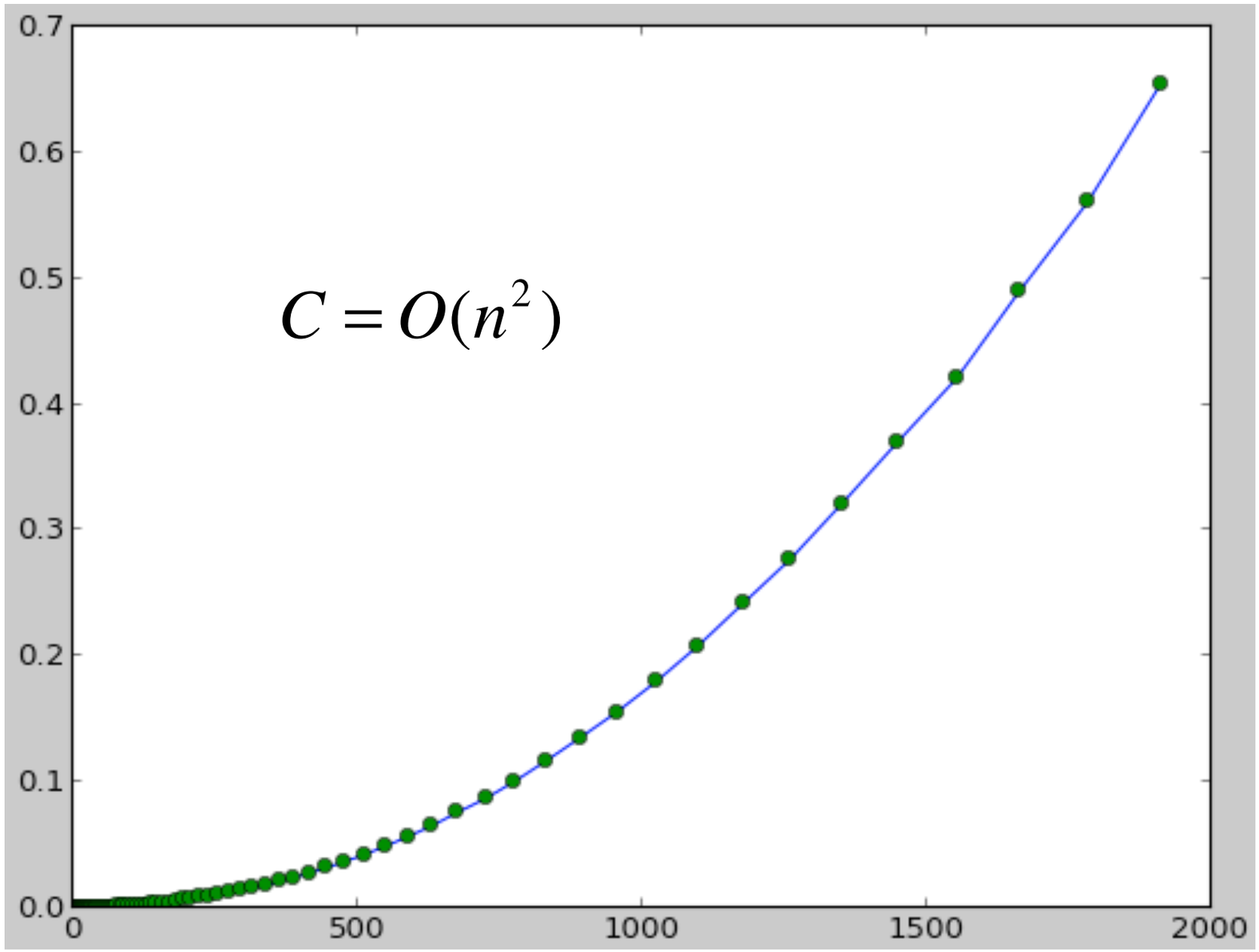
- L'algorithme «coûte» plus ou moins de **mémoire**. [stockage des données]
- L'algorithme «coûte» plus ou moins de **temps**.

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

On constate que le temps nécessaire pour N=2000 et 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 :

Temps (s)



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 ?

```
1  def algo(N):
2      for i in range(2,N):
3          PROP=True
4          for j in range(2,i):
5              if (i%j==0):
6                  PROP=False
7          if (PROP):
8              print("{:5d}".format(i), end='')
9
10 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 ....
```

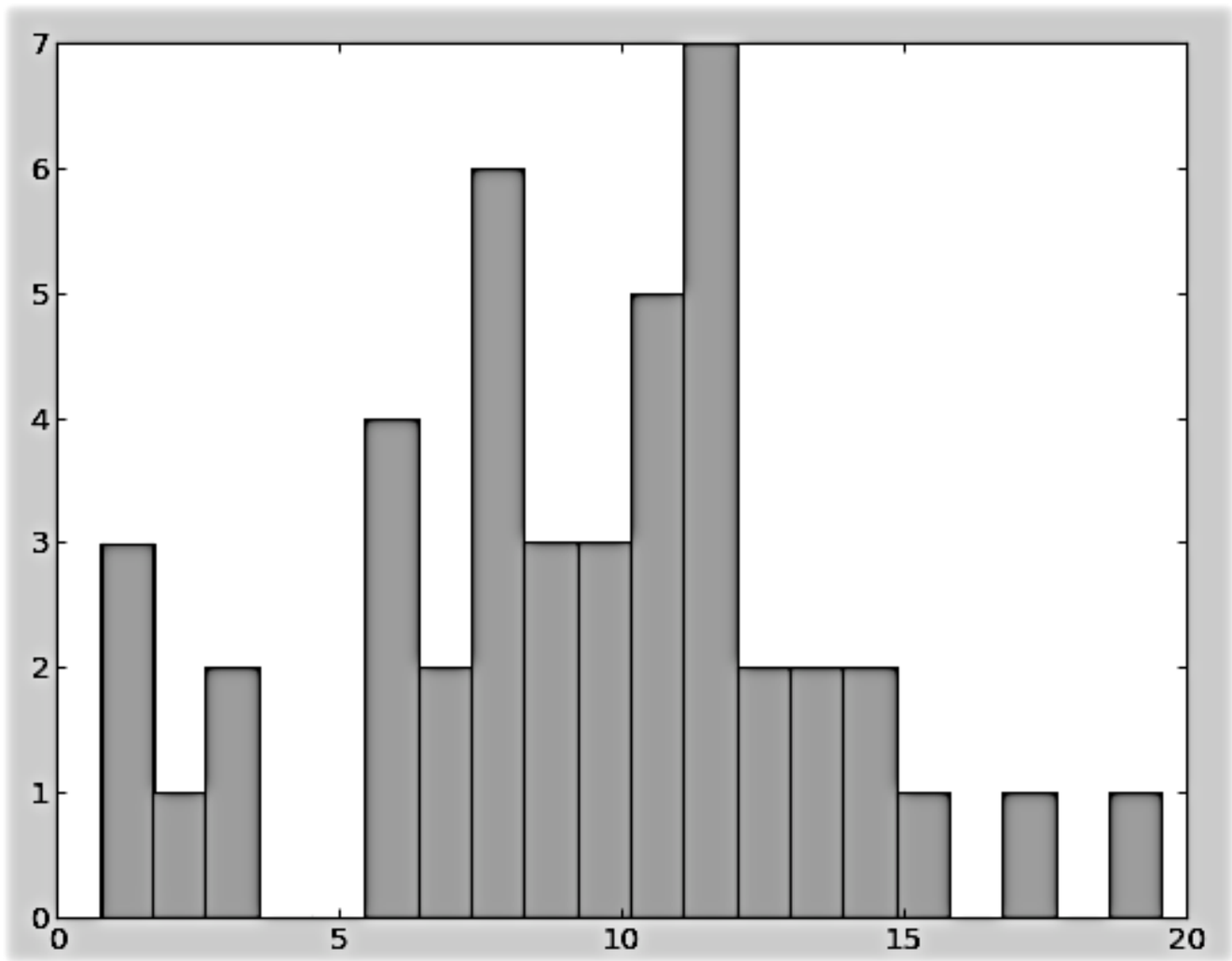
2 - Etude d'un tableau de réels

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
```

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 ?

Itération :

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)
```

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.

Démonstration et terminaison

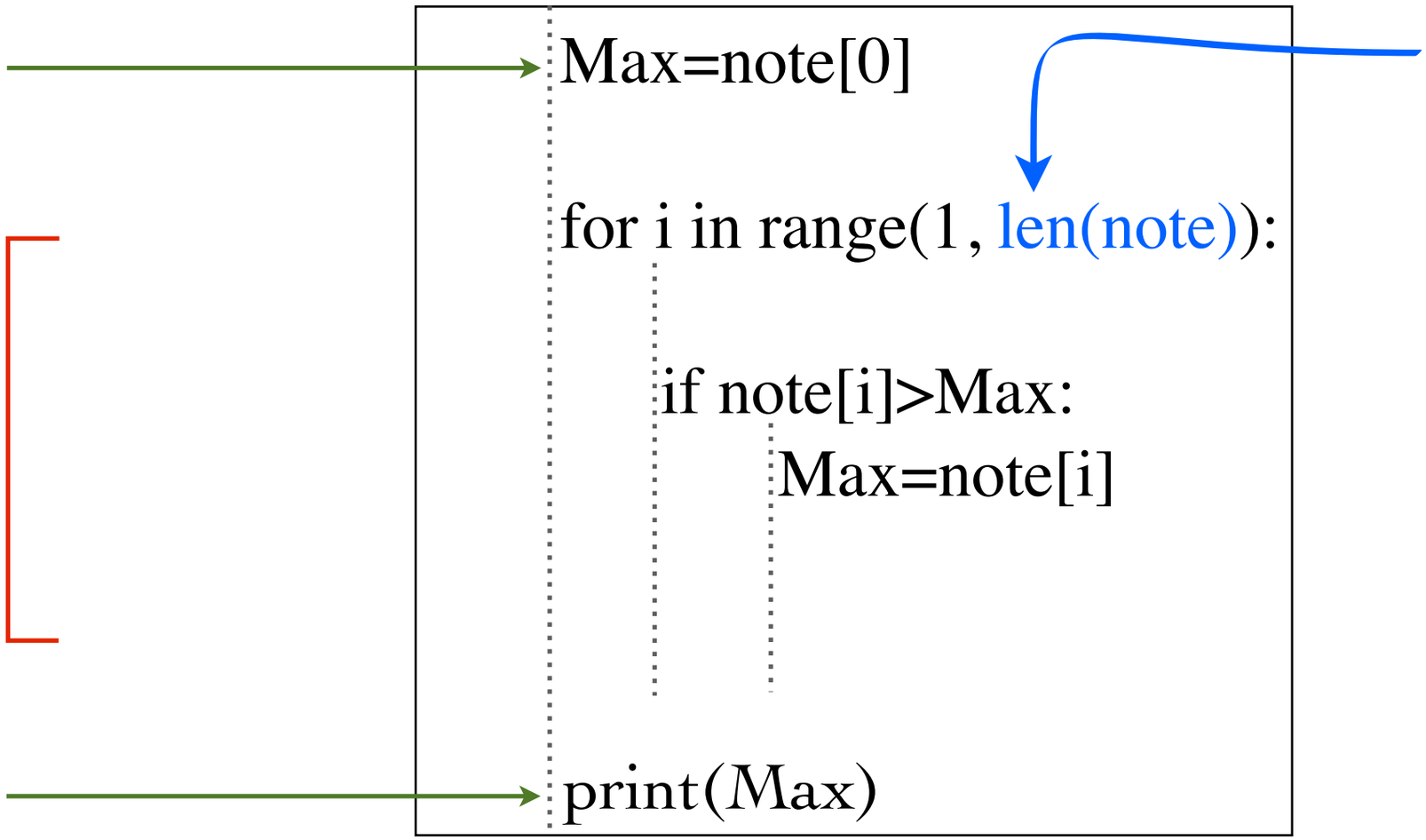
Invariant de boucle :

Initialisation :

Itération :

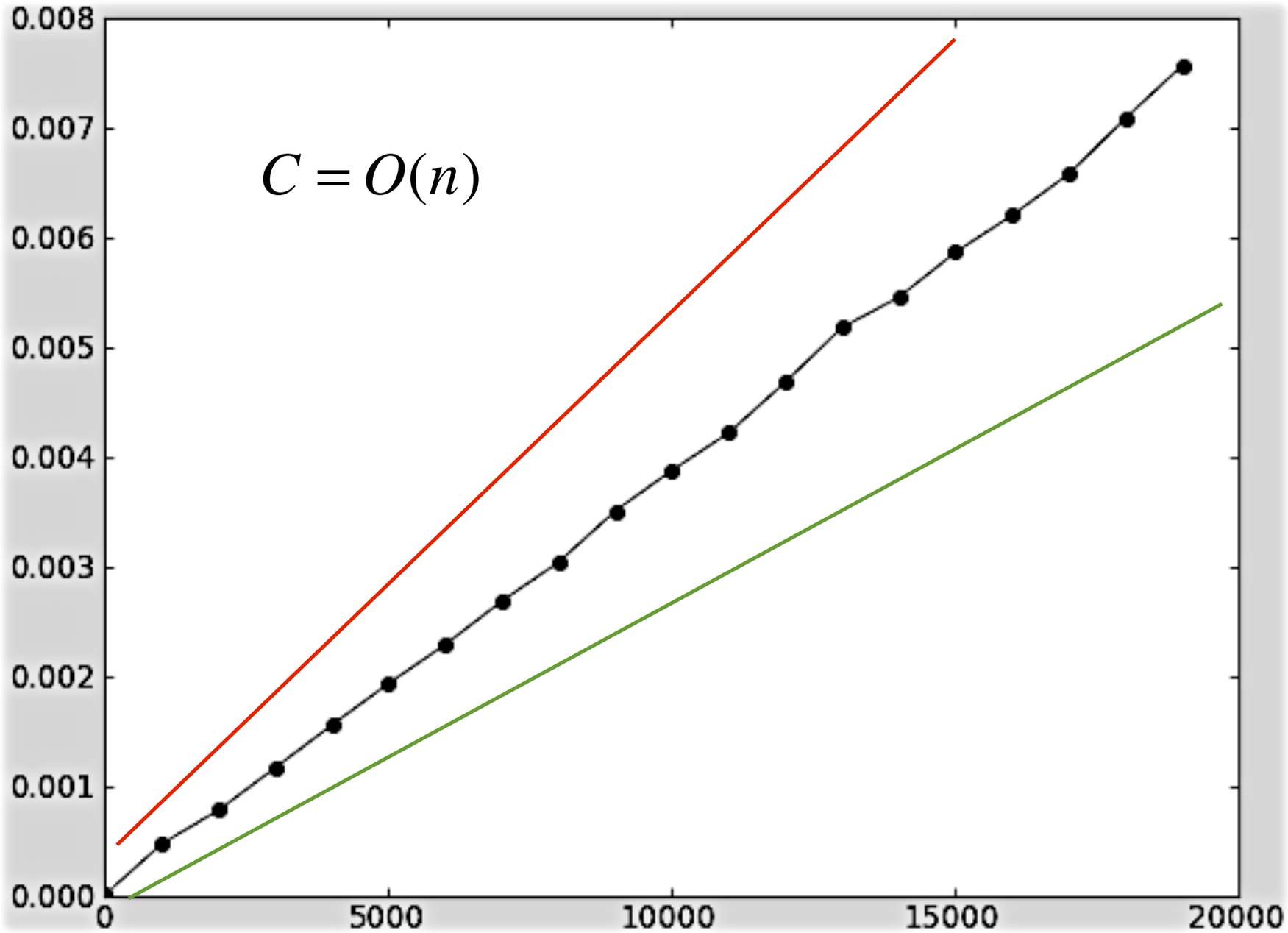
Conclusion :

Comment mesurer le coût en temps ?



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

Temps (s)



N

<u>Complexité linéaire :</u>

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

- 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.

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»

Information : «plus petit» ou «plus grand»

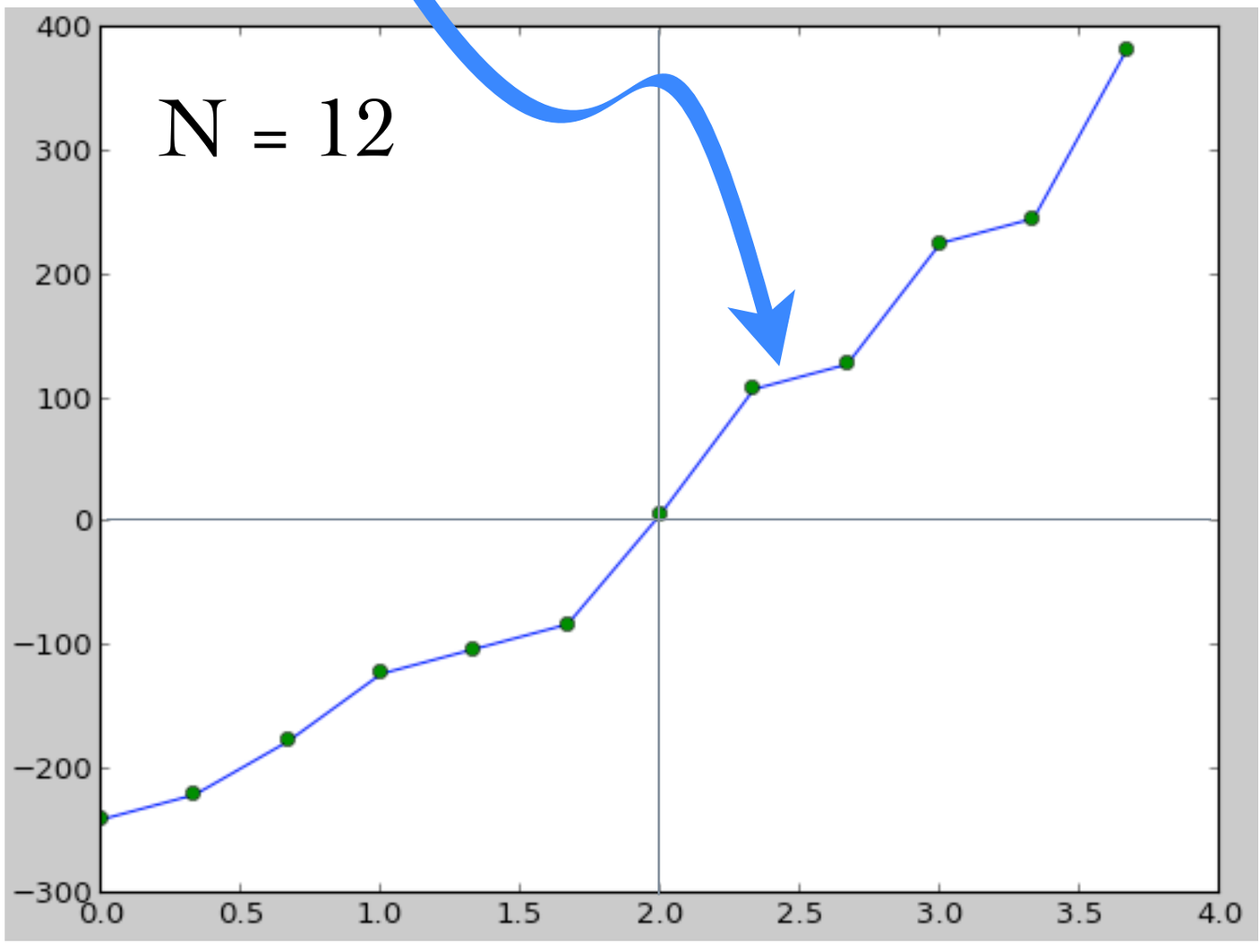
Exercice : Trouver deux algorithmes possibles

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



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.

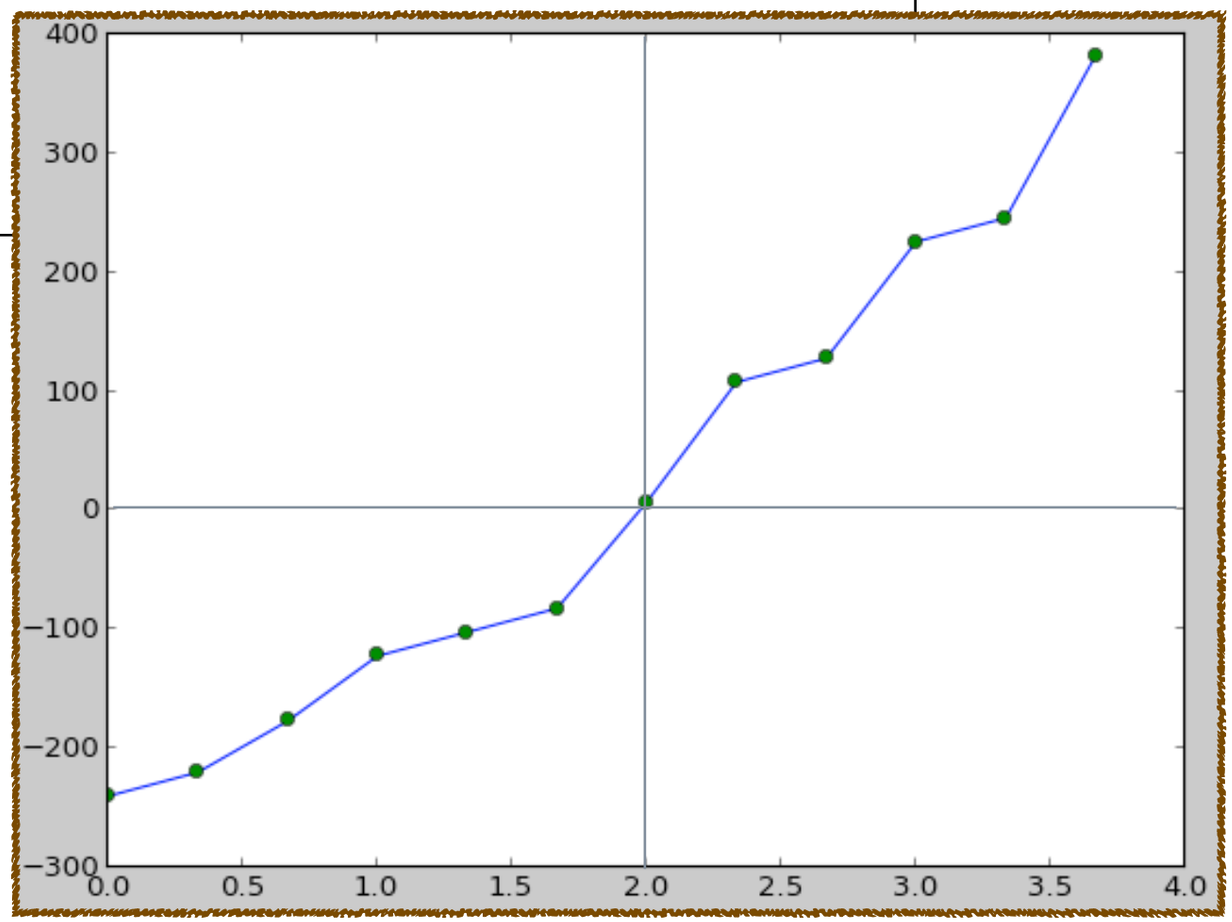
=> la fonction nous renvoie également l'intervalle des x où la fonction s'annulle.
On en déduira un encadrement sur x de la position du zéro !

```
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 :

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) regénère 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()
```



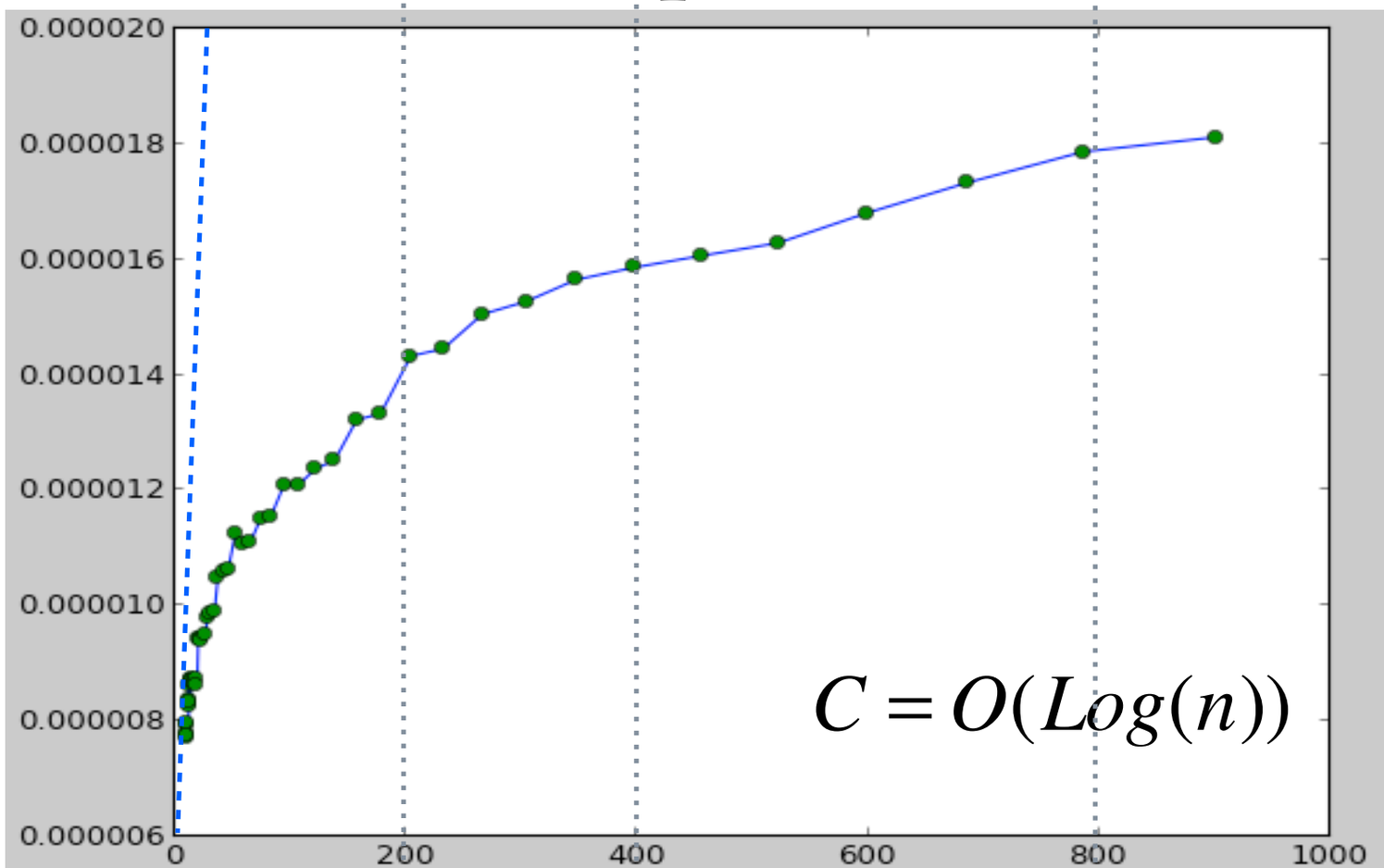
Algorithmes Naïf de recherche du zéro de la fonction

Boucle sur Tab : **N - fois**

```
#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]);
```

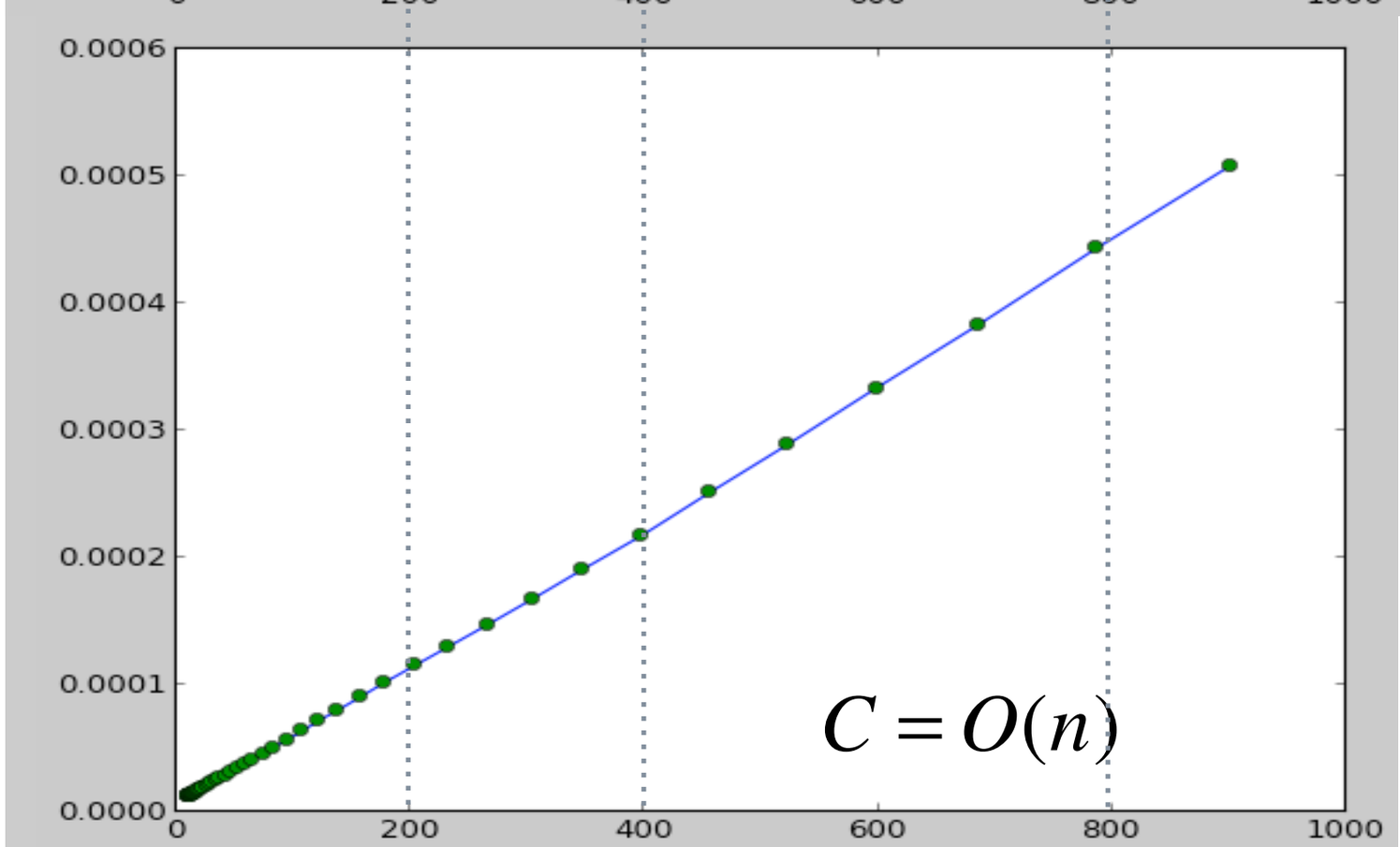
Chaque point est une expérience aléatoire répétée 1000 fois !

Dichotomie



Naïf

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



Algorithmes de recherche du zéro par dichotomie

Boucle sur Tab : **? - fois**

(même temps pour chaque itération)

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

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

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

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

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

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

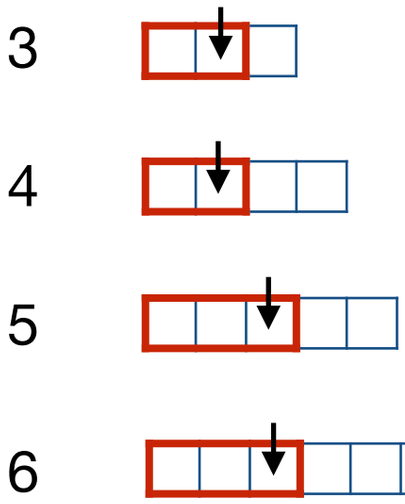
Si n ⩽ 2 on a trouvé le zéro.

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 :

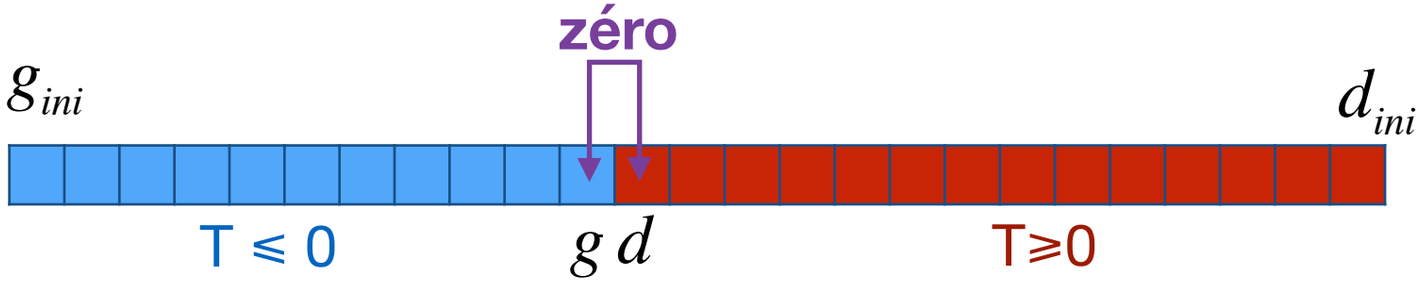
$$\mathbb{C} = a_1 + a_2 + k(b_1 + (b_2) + (b_3)) + a_3 + a_4$$

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

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

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 !

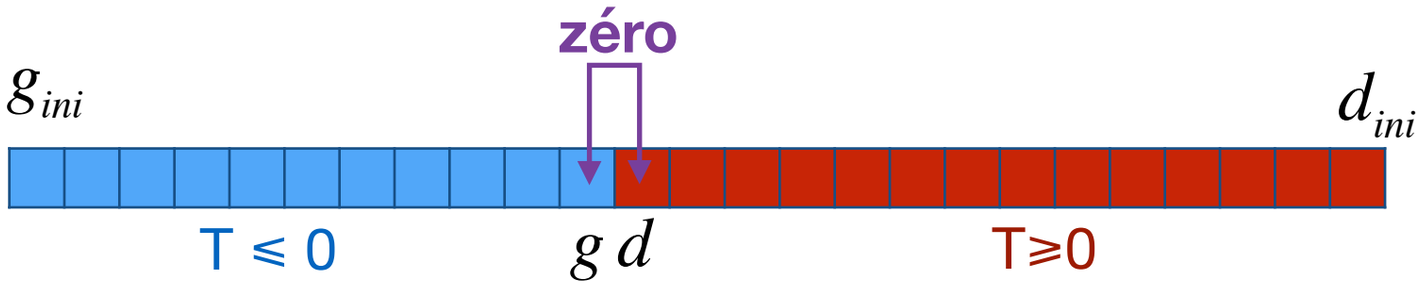
- Le zéro est dans $T[g_{new} : d_{new} + 1]$
- $g_{new} \leq \frac{g+d}{2} < \frac{d-1+d}{2} = \frac{2d-1}{2} < d = d_{new}$ $g_{new} < d_{new}$

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)

<u>Condition d'arrêt :</u> **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 !