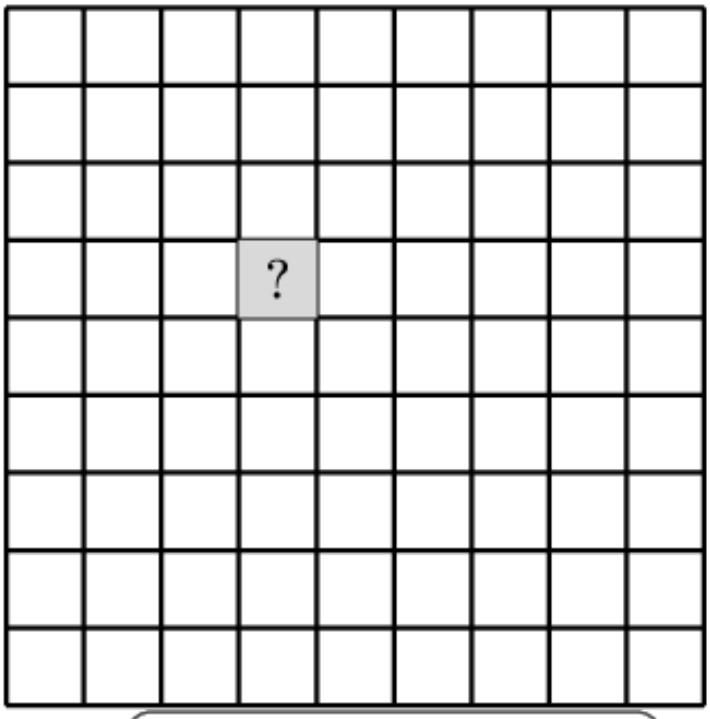
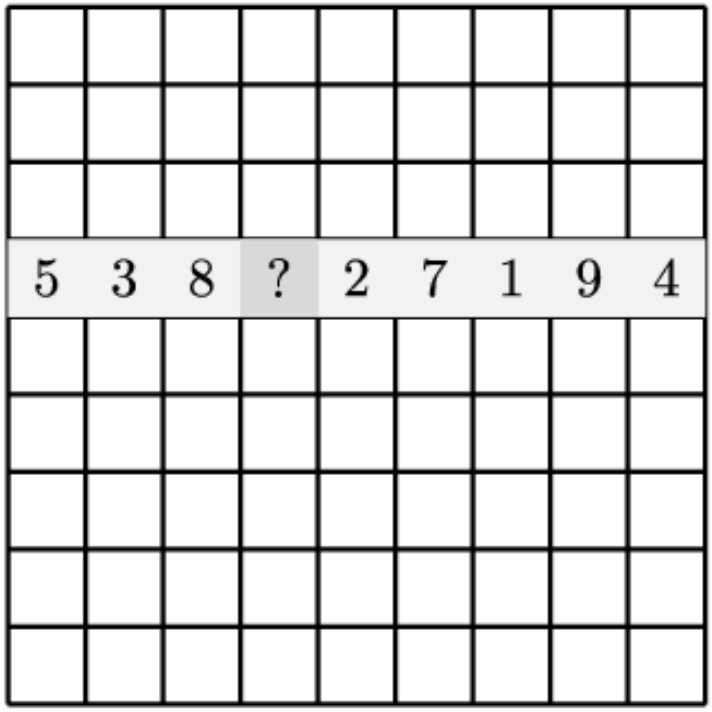
new idea

# SOLUTION

Sudoku




**Initial Candidates:**
$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

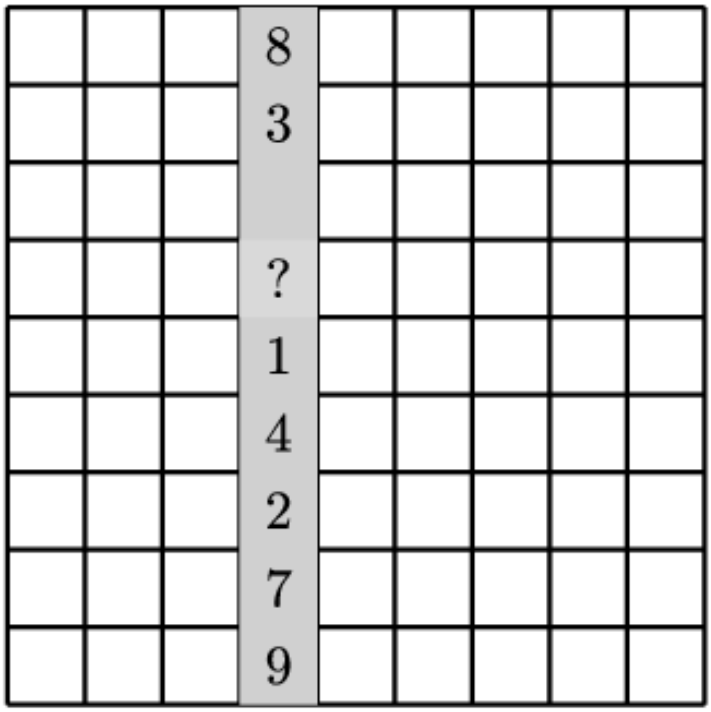



**After discard :**

$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$\downarrow$

$\{\mathbf{6}\}$

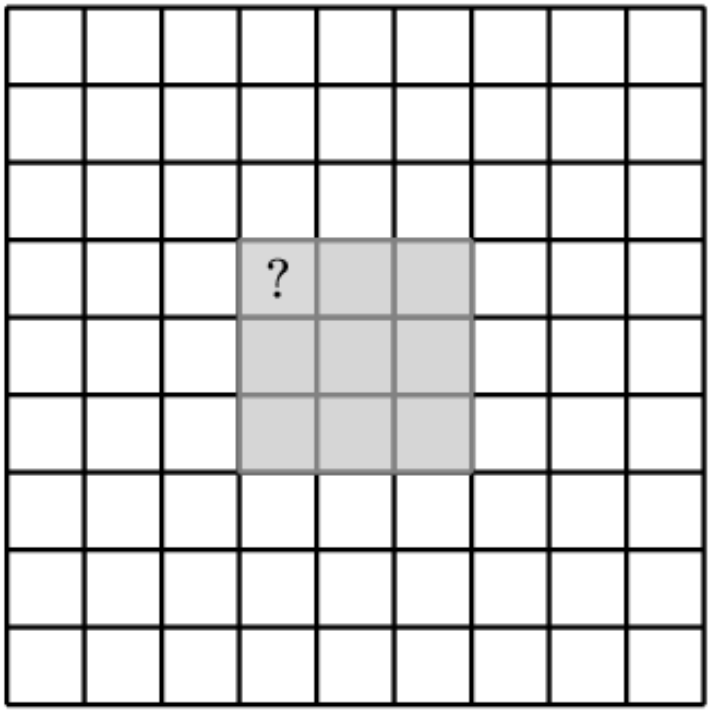



**After discard :**

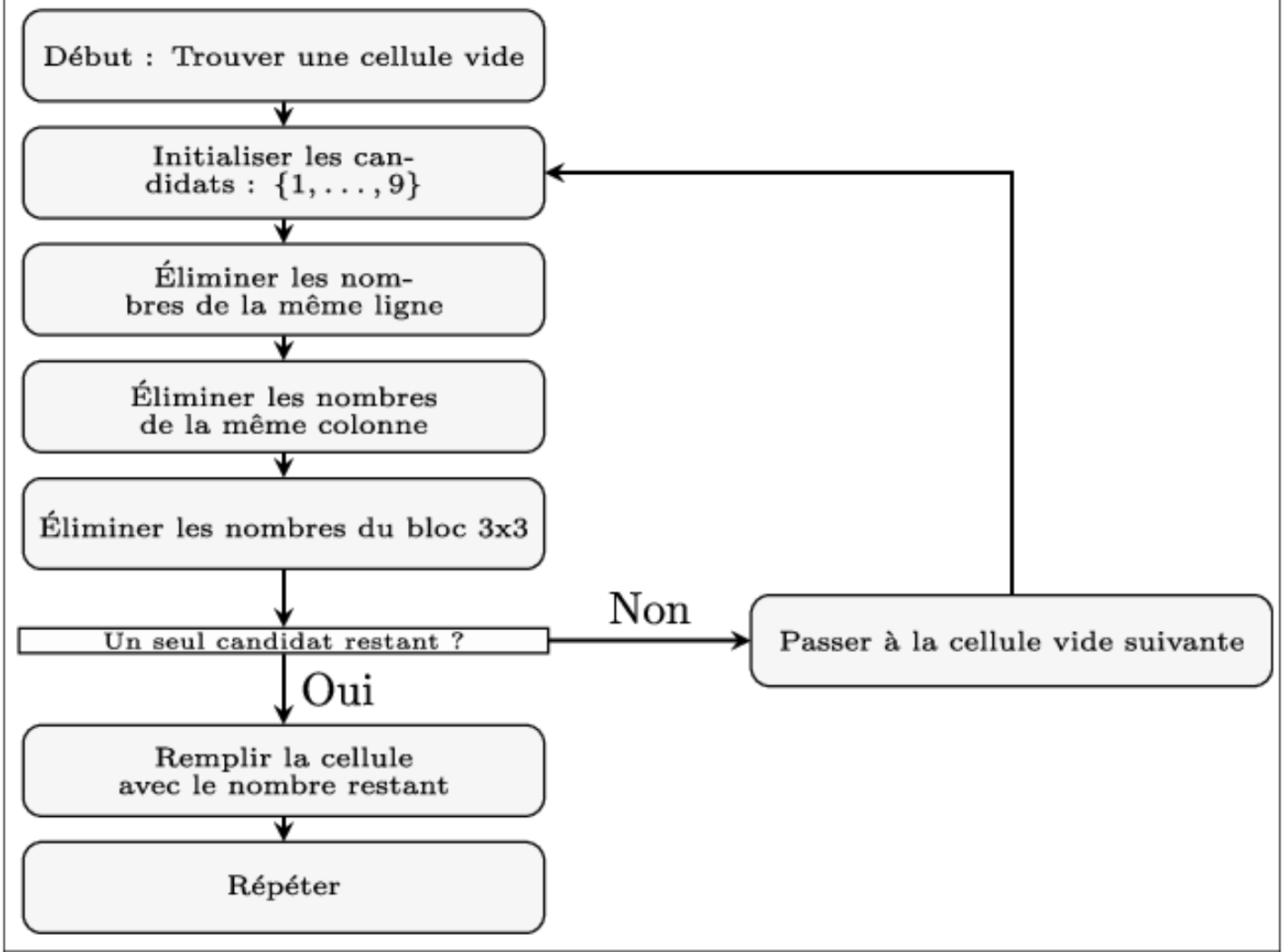
$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

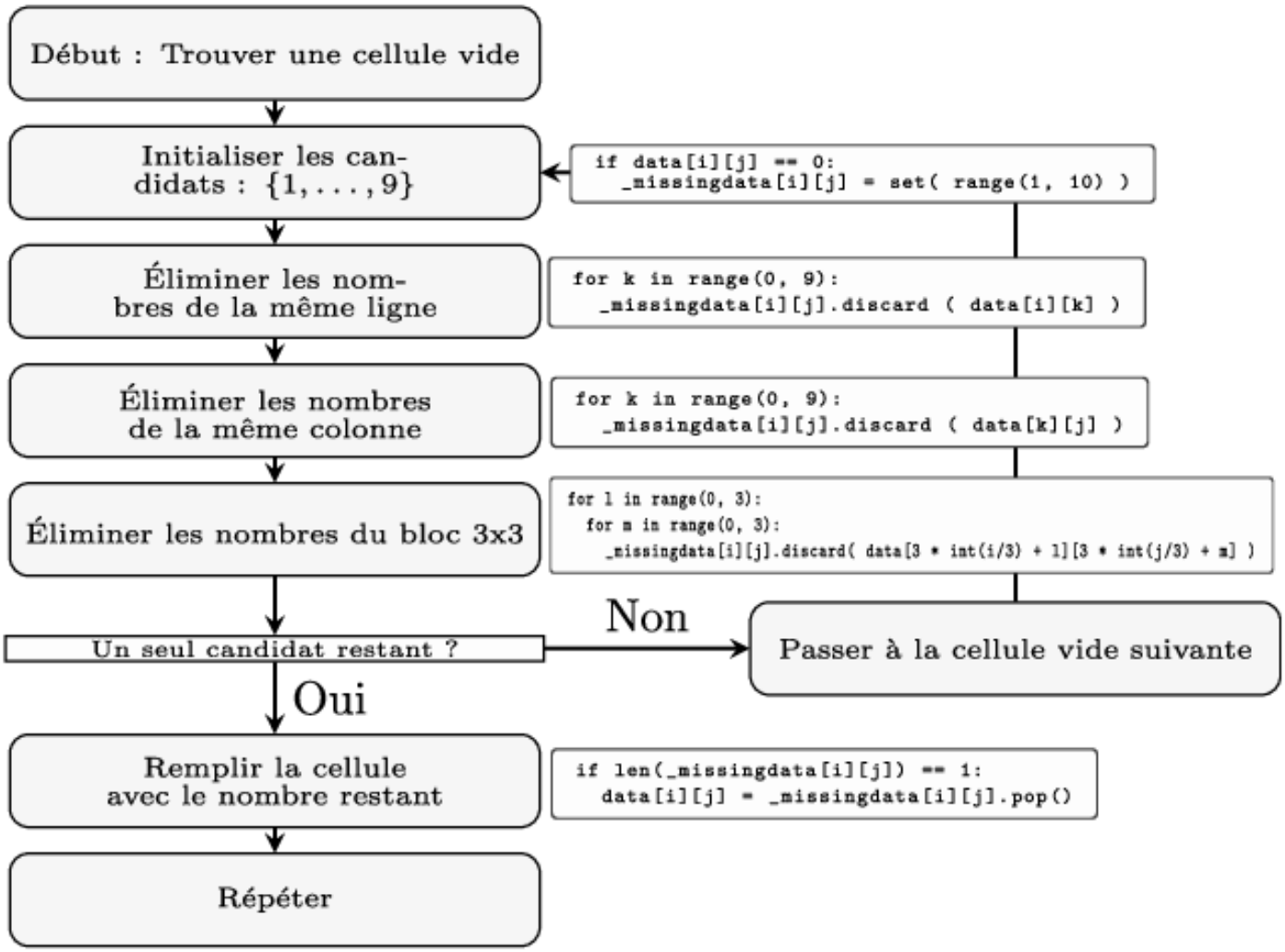
$\downarrow$

$\{\mathbf{5}, \mathbf{6}\}$




**After discard:**
If only one candidate remains → **Fill!**






```
def solveMissingPuzzle(data):
  _missingdata = [ [] ]*(9)
  for i in range(0, 9):
    _missingdata[i] = [ [] ]*(9)
    for j in range(0, 9):
      if data[i][j] == 0:
        _missingdata[i][j] = set( range(1, 10) )

    for i in range(0, 9):
      for j in range(0, 9):
        if data[i][j] == 0:
          for k in range(0, 9):
            _missingdata[i][j].discard ( data[k][j] )
            _missingdata[i][j].discard ( data[i][k] )

          for l in range(0, 3):
            for m in range(0, 3):
              _missingdata[i][j].discard(
                   data[3 * int(i/3) + l][3 * int(j/3) + m] )

          if len(_missingdata[i][j]) == 1:
            data[i][j] = _missingdata[i][j].pop()
```

new idea

# Asymptotic Analysis

```
n = 100

sum = 0
for i in range(0, n):
    sum += i
print("sum is: ", sum)
```

```
4950
```

la boucle s'exécute sur 100 étapes

mais quel est le temps d'exécution de l'ordinateur ?

```
n = 100

sum = 0
for i in range(0, n):
    sum += i
print("sum is: ", sum)
```

la réponse *exacte* dépend de beaucoup d'autres choses

cpu speed

programming language

optimization level

système d'exploitation

```
n = 100 une constante

sum = 0 une constante
for i in range(0, 100):
    sum += i une constante
print("sum is: ", sum)
                  une constante
```

La réponse *approximative* est beaucoup plus simple :

pour $n = 100$, c'est $\approx c \cdot 100$, où $c \geq 1$ est une constante

en général, c'est $\approx c \cdot n$, où $c \geq 1$ est une constante

'masque' tous les nombres liés à l'ordinateur dans la constante $c$

**s'intéresse principalement: comment le temps d'exécution change avec $n$**

```
import time
n = pow(10,5)
start_time = time.time()
sum = 0
for i in range(0, n):
    sum += i
end_time = time.time()
print("time: ", end_time - start_time)
```

```
time:  0.0070209503173828125
```

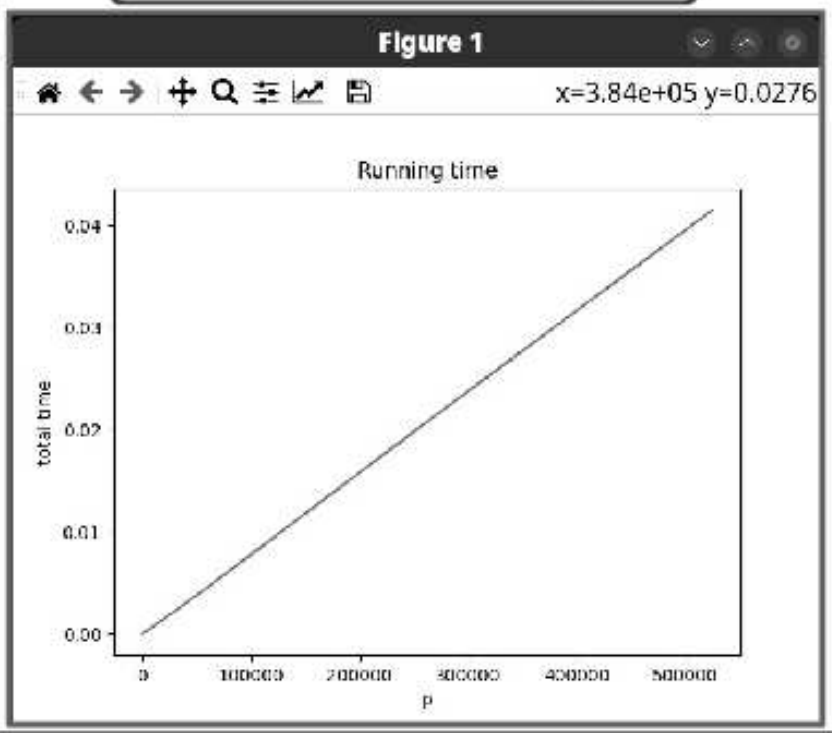
```
import time
n = pow(10,6)
start_time = time.time()
sum = 0
for i in range(0, n):
    sum += i
end_time = time.time()
print("time: ", end_time - start_time)
```

```
time:  0.05660414695739746
```

```
import time
n = pow(10,7)
start_time = time.time()
sum = 0
for i in range(0, n):
    sum += i
end_time = time.time()
print("time: ", end_time - start_time)
```

```
time:  0.5137147903442383
```

```
import time
for p in range(1, 100):
    n = pow(10,p)
    start_time = time.time()
    sum = 0
    for i in range(0, n):
        sum += i
    end_time = time.time()
```



## Principaux principes philosophiques :

**1.** nous nous intéressons à la façon dont le temps varie avec un grand $n$

**2.** les constantes multiplicatives ne sont pas importantes (pour l'instant !)

Nous avons besoin d'une définition de '$\leq$' avec ces principes

---

**Algorithm 1:** le temps d'exécution sur une input de taille $n$: $f(n)$

**Algorithm 2:** le temps d'exécution sur une input de taille $n$: $g(n)$

On dit que **Algorithm 1** n'est pas plus lent que **Algorithm 2** si

- pour tout $n$ suffisamment grand—$n \geq a$, ou a est un constante
- il existe une constante $c$

tel que

$$f(n) \leq c \cdot g(n)$$

Nous écrivons cela sous la forme

$$f(n) = O\Big(g(n)\Big)$$

Pour montrer que $f(n) = O\big(g(n)\big)$, il faut montrer qu'il existe

- une valeur de $a$
- une valeur de $c$

telles que pour tout $n \geq a$, on a

$$f(n) \leq c \cdot g(n)$$

**Question:** $100n = O(n)$ ?

**Yes**

- $a = 1$
- $c = 100$

pour tout $n \geq 1$, on a $\quad 100\,n \leq 100 \cdot n$

**Question:** $n^2 + n = O\left(n^2\right)$ ?

**Yes**

- $a = 1$
- $c = 2$

pour tout $n \geq 1$, on a $\quad n^2 + n \ \leq \ 2 \cdot n^2$

---

**Question:** $n \log_2 n = O\left(n\right)$ ?

**No**

pas possible de trouver des constantes a et c t.q.

pour tout $n \geq a$, on a

$n \log_2 n \leq c \cdot n \quad \Longrightarrow \ \log_2 n \leq c$

**donc c augmente avec $n$, ce qui n'est pas autorisé.**

**Cela donne également une notion d'égalité approximative :**

$$\mathbf{f(n) = \Theta(g(n))}$$

ssi $f(n) = O(g(n))$ **et** $g(n) = O(f(n))$.

**Exemples où c'est $\Theta$ :**

$5n^2 + 3n = \Theta(n^2)$

$2^n + n^{100} = \Theta(2^n)$

$n \log n + n = \Theta(n \log n)$

**Exemples où ce n'est pas $\Theta$ :**

$n^2 \neq \Theta(n^3)$

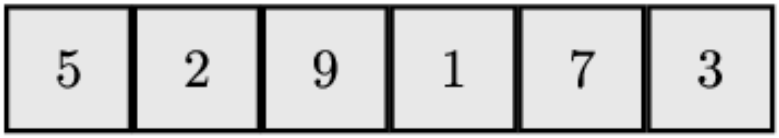
$\log n \neq \Theta(n)$

$n \neq \Theta(n \log n)$

new idea

# Divide and Conquer, I

**Problem 1:** étant donné une liste non triée `A` de `n` entiers, et un entier `x`, nous voulons savoir si `x` est dans `A`.




```
A = [5, 2, 9, 1, 7, 3]
```

```
x = 9
```

Il existe un algorithme simple qui est le meilleur possible.

$\Longrightarrow$ parcourir `A` en boucle et de vérifier chaque élément par rapport à `x`.

```python
def find(A, x):
  n = len(A)

  for i in range(0, n):
    if x == A[i]:
      return True

  return False
```

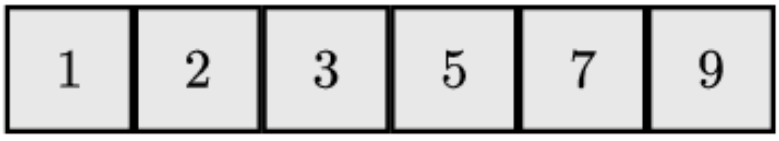
Temps pris : `n` itérations de la boucle, donc O(`n`).

Il n'est pas difficile de se convaincre que l'on ne peut pas faire mieux !

new idea

# BINARY SEARCH

**Problème 2 :** Étant donné une liste *triée* `A` de `n` entiers, et un entier `x`, nous voulons savoir si `x` est dans `A`.




```
A = [1, 2, 3, 5, 7, 9]
```

```
x = 9
```

Bien sûr, comme précédemment, nous pouvons utiliser une boucle pour résoudre ce problème, en temps O(**n**).

Mais le fait que `A` soit triée peut-il être utilisé pour faire mieux ?

Oui !

**Binary Search, $x = 21$ — Step 1**

low index = 0, high index = 15

mid index = (0+15) // 2 = 7


$15 < 21$

→ **search right half**

**Binary Search, $x = 21$ — Step 2**

low index = 8, high index = 16

mid index = $(8+16)/2 = 12$


$23 > 21$

$\rightarrow$ **search left half**

**Binary Search, $x = 21$, — Step 3**

low index = 9, high index = 12

mid index = $(9+12)/2 = 10$


$19 > 21$

$\rightarrow$ **search right half**

**Binary Search, x = 21, — Step 4**

low index = high index = 11

Found!


x = **21** found at position 11

---

**Entrée:** Une liste triée `A` de `n` entiers, un entier `x`

**Sortie :** True si `x` $\in$ `A`, False sinon

Vérifier l'élément du milieu de `A` ;

**si** *A[n//2]* $>$ *x* **alors si**

| `x` se trouve dans `A`, il doit être parmi les premiers `n//2` éléments de `A`

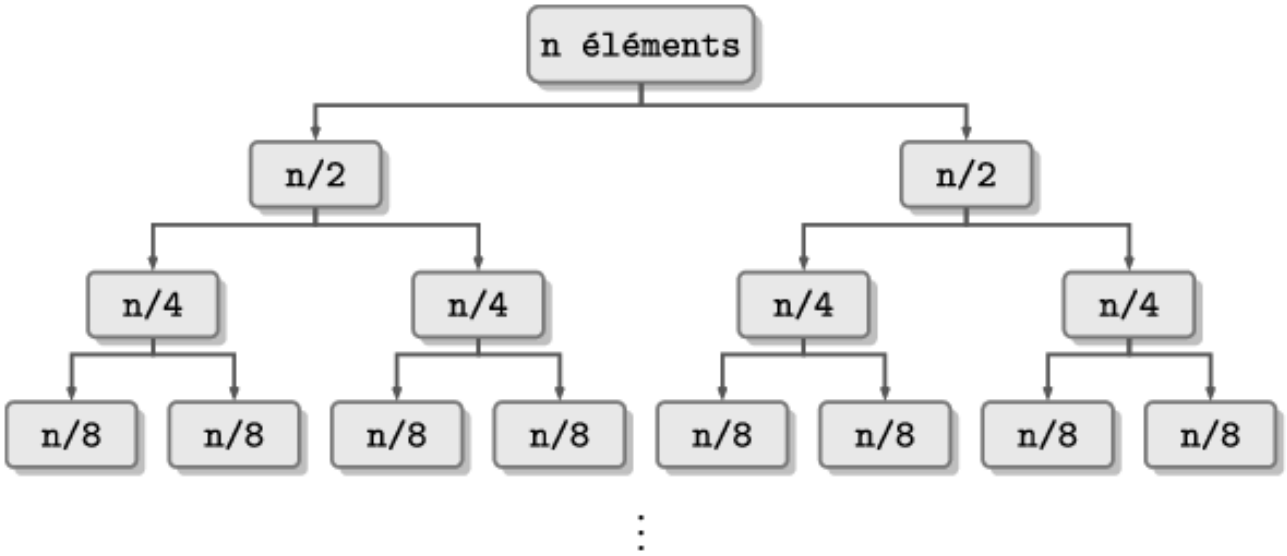
**sinon** *A[n//2]* $<$ *x* **alors si**

| `x` se trouve dans `A`, il doit être parmi les derniers `n//2` éléments de `A`

**sinon** *x est égal à l'élément du milieu* **alors si**

| nous avons trouvé `x` !

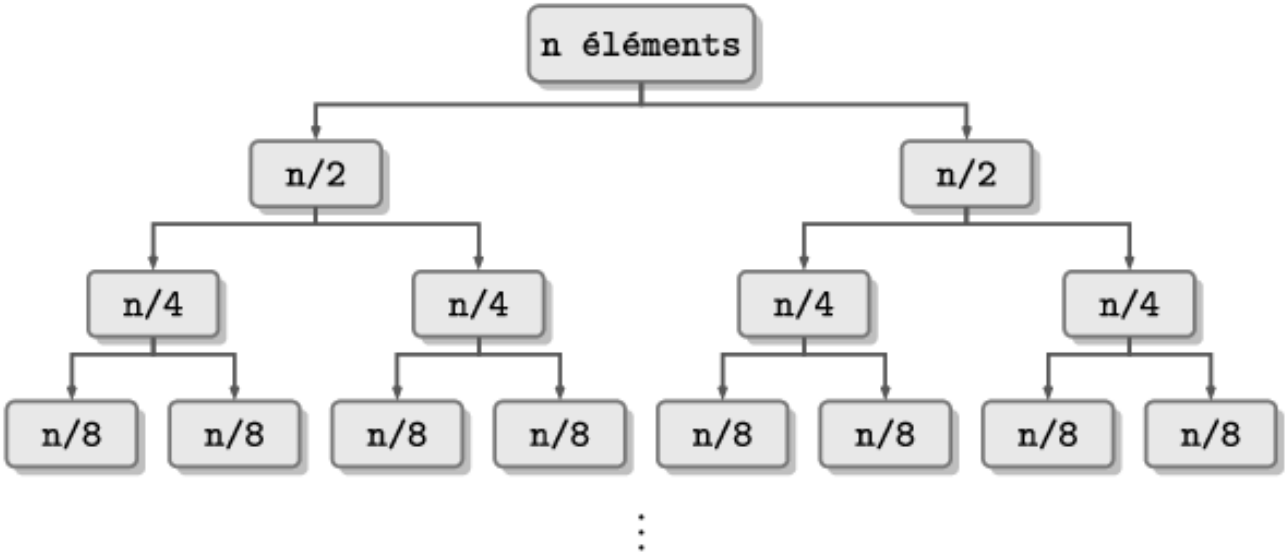
---




Après $i$ étapes, la sous-liste a une taille de $\frac{n}{2^i}$.

Dans le pire des cas, nous continuons jusqu'à ce que cette taille devienne 1.

Ainsi, nous devons continuer jusqu'à ce que $\frac{n}{2^i} \leq 1$.




$$\frac{n}{2^i} \leq 1 \iff n \leq 2^i \iff \log_2 n \leq i$$

Ainsi, après $\log_2 n$ étapes, notre sous-liste a une taille de 1 !

## Que s'est-il vraiment passé ?

Au début, nous avions $n$ éléments.

Après 1 étape, il nous restait $n/2$ éléments.

⋮

Après $i$ étapes, il nous restait $n / 2^i$ éléments.

Donc la taille du problème diminue très rapidement !

Un calcul montre que nous avons terminé après $\log_2 n$ étapes.

Cette idée s'appelle **Diviser pour régner**.

## Iterative view

```
def binarySearch(A, x):
    start, end = 0, len(A)-1

    while end >= start:
        mid = (start+end)//2
        if x == A[mid]:
            return mid
        if x > A[mid]:
            start = mid+1
        if x < A[mid]:
            end = mid-1
    return False

import random
A = sorted( [ random.randint(0,20) for _ in range(0,10) ] )
print(A)

x = 4
print(f Is {x} in A:   , binarySearch(A, x))
```

## Recursive view

Une façon équivalente : une fonction récursive !

```
def binarySearch_RECURSIVE(A, start, end, x):
    if end < start:
        return None

    mid = (start+end) // 2
    if x == A[mid]:
        return mid
    if x < A[mid]:
        return binarySearch_RECURSIVE(A, start, mid-1, x)
    else:
        return binarySearch_RECURSIVE(A, mid+1, end, x)

import random
A = sorted( [ random.randint(0,20) for _ in range(0,10) ] )
print(A)

x = 4
print(f Is {x} in A: ,binarySearch_RECURSIVE(A,0,len(A)-1, x))
```

## Utiliser in vs Recherche Binaire

```
import time, random, math

def binarySearch(A, x):
    start, end = 0, len(A)-1

    while (end-start)>=0:
        mid = (start+end) // 2
        if x == A[mid]:
            return mid
        elif x > A[mid]:
            start = mid+1
        elif x < A[mid]:
            end = mid-1
    return -1

n = 100000000
A = sorted([ random.randint(0, 20) for _ in range(0, n) ])

start = time.time()
ans = binarySearch(A, 6)
binarySearch_time = time.time() - start

start = time.time()
ans3 = 6 in A
IN_time = time.time() - start

print(f Time for binarySearch: {binarySearch_time:f} )
print(f Time for IN: {IN_time:f} )
```

```
Time for binarySearch: 0.000012
Time for IN: 0.121071
```

≈ 10000 fois plus rapide

# new idea

# COMPUTING sqrt

Application : Racine carrée d'un nombre

**Problème :** Étant donné un nombre $S \geq 1$, trouver la partie entière de $\sqrt{S}$.

Exemple: pour $S = 22$ :

$$1^2 = 1 < 22$$
$$10^2 = 100 > 22$$

$\Longrightarrow$ on peut deduire que la solution est entre 1 et 10 !

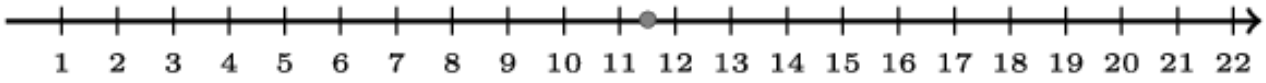
**Algorithme :** Recherche binaire sur l'intervalle $[1, S]$

- À chaque étape, on regarde le milieu $m$
- Si $m^2 \leq S \quad \Longrightarrow \quad$ on cherche à droite
- Sinon $\quad \Longrightarrow \quad$ on cherche à gauche

**Temps :** $O(\log S)$ étapes avant que la longueur de l'intervalle devienne $< 1$
Une fois que cela se produit, il est facile de trouver la solution.

# S = 22

$$m = \frac{(1+22)}{2} = 11.5$$

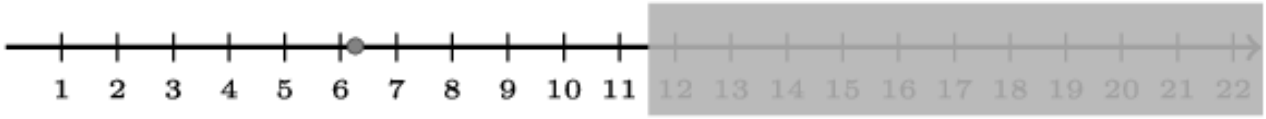



$$11.5^2 = 132.25 > 22 \qquad \Longrightarrow \qquad m \textbf{ is too big}$$

**rechercher dans l'intervalle** $[1, 11.5]$

# S = 22

$$m = \frac{(1+11.5)}{2} = 6.25$$




$$6.25^2 = 39.0625 > 22 \qquad \Longrightarrow \qquad m \textbf{ is too big}$$

**rechercher dans l'intervalle** $[1, 6.25]$

# S = 22

$$m = \frac{(1 + 6.25)}{2} = 3.625$$


$$3.625^2 = 13.140625 < 22 \qquad \Longrightarrow \qquad m \textbf{ is too small}$$

**rechercher dans l'intervalle** $[3.625, 6.25]$

# S = 22

$$m = \frac{(3.625 + 6.25)}{2} = 4.9375$$


$$4.9375^2 = 24.37890625 > 22 \qquad \Longrightarrow \qquad m \textbf{ is too big}$$

**rechercher dans l'intervalle** $[3.625, 4.9375]$

# S = 22

$$m = \frac{(3.625 + 4.9375)}{2} = 4.28125$$


$4.28125^2 = 18.329101562 < 22 \quad \Longrightarrow \quad m$ **is too small**

**rechercher dans l'intervalle** $[4.28125, 4.9375]$

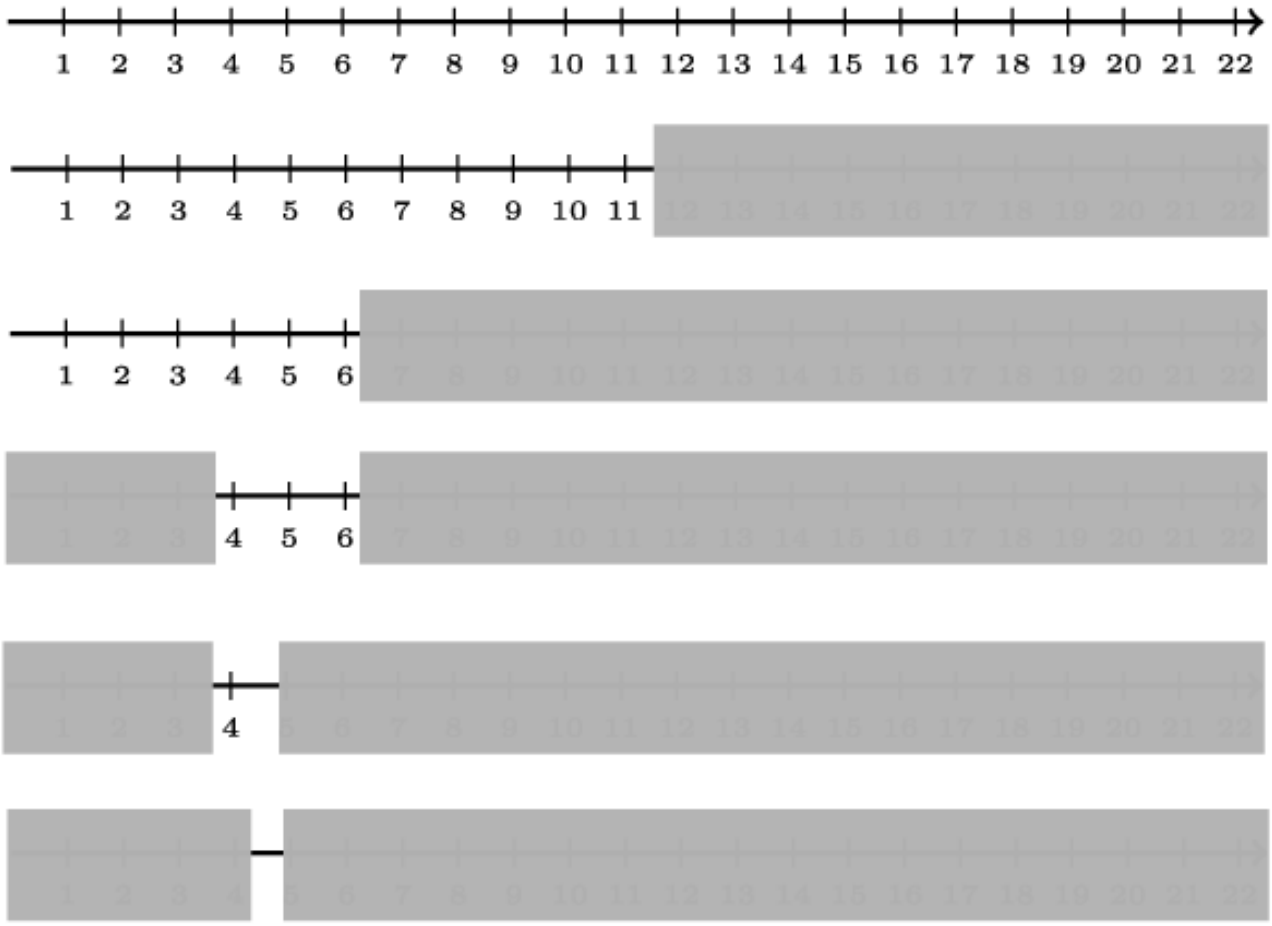
# S = 22

Donc $\sqrt{22}$ se situe entre $4.28125$ et $4.9375$.


**nous arrêtons puisque l'intervalle** $[4.28125, 4.9375]$ **a longeur** $< 1$

$\Longrightarrow$ la solution est 4.

Notre chemin




```python
import math

def mysqrt(S):
    start = 1
    end = S

    for i in range(0, int(math.log2(S)) + 1):
        m = (start + end) / 2.0

        if m * m == S:
            return int(m)
        elif m * m < S:
            start = m
        else:
            end = m

    if ( int(end)*int(end) > S ):
        return int(start)
    else:
        return int(end)

print( mysqrt(22) )
```

4

```python
def mysqrt(x, k):
    start = 1
    end = x

    for i in range(0, k):
        m = (start+end)/2.0

        if x == m*m:
            return m
        elif x < m*m:
            end = m
        elif x > m*m:
            start = m

    return (start+end)/2.0

for i in range(1, 10):
    print(f k={i}: {mysqrt(22, i)} )
```

```
k=1: 6.25
k=2: 3.625
k=3: 4.9375
k=4: 4.28125
k=5: 4.609375
k=6: 4.7734375
k=7: 4.69140625
k=8: 4.650390625
k=9: 4.6708984375
```

new idea

# WOOD CUTTING

Application : Coupe de bois (problème d'optimisation)

**Problème :** On a `n` morceaux de bois de longueurs différentes.
On veut obtenir au moins `k` morceaux de même longueur `L`, ou `L` est un entier.

**Question :** Quel est le `L` maximal possible ?

**Exemple :** Bois = [20, 15, 10, 17], `k` = 7

Avec `L` = 5 : on obtient 4 + 3 + 2 + 3 = 12 pièces ($\geq 7$)

Avec `L` = 8 : on obtient 2 + 1 + 1 + 2 = 6 pièces ($< 7$)

La réponse est entre 5 et 8 !

**Idée clé :** Si `L` est réalisable, alors tout `L'` $\leq$ `L` l'est aussi.
La fonction "nombre de pièces" est *décroissante* en `L`.

**Algorithme :** Recherche binaire sur `L` dans $[1, \max(\text{bois})]$

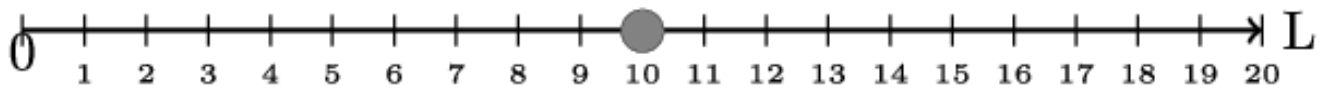
- À chaque étape, on vérifie si `L` donne $\geq$ `k` pièces
- Si oui, on cherche plus grand (`L` $\leftarrow$ m + 1)
- Sinon, on cherche plus petit (`R` $\leftarrow$ m - 1)

**Complexité :** $O(\texttt{n} \cdot \log(\max(\text{bois})))$

---

$$\texttt{Wood} = [20,15,10,17], \quad k = 7$$

$$L = (0 + 20)/2 = 10$$

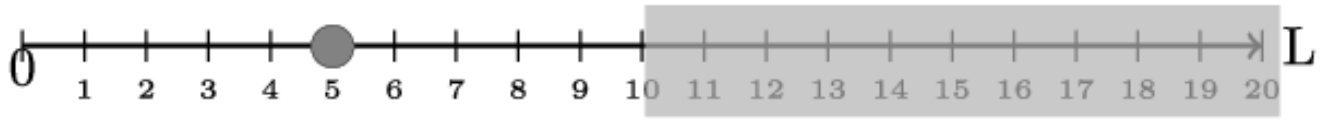



**Nombres des pieces pour** $L = 10 : 2 + 1 + 1 + 1 = 5$

$$5 < 7 \quad \Longrightarrow \quad L \textbf{ is too big}$$

`Wood = [20,15,10,17],` $k = 7$

$L = (0 + 10)/2 = 5$

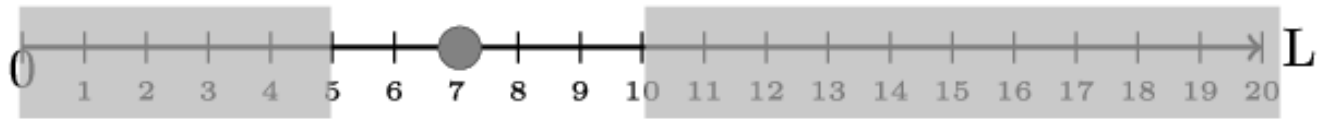



**Nombres des pieces pour** $L = 10 : 4 + 3 + 2 + 3 = 12$

$12 > 7 \quad \Longrightarrow \quad$ **$L$ is too small**

`Wood = [20,15,10,17],` $k = 7$

$L = (5 + 10)//2 = 7$

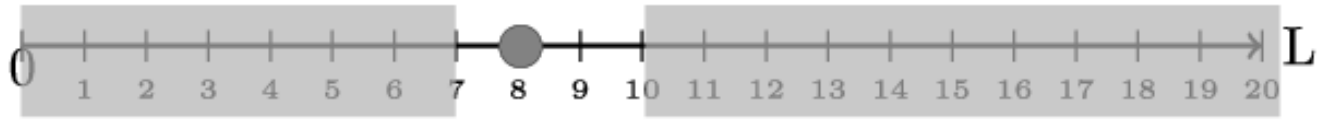



**Nombres des pieces pour** $L = 7 : 2 + 2 + 1 + 2 = 7$

$7 \geq 7 \quad \Longrightarrow \quad$ **$L$ is too small**

`Wood = [20,15,10,17],` $k = 7$

$$L = (7 + 10)/2 = 8$$




**Nombres des pieces pour** $L = 8$**:** $2 + 1 + 1 + 2 = 6$

$6 < 7 \quad \Longrightarrow \quad L$ **is too big**

So the final answer is $L = 7$!

`Wood = [20,15,10,17],` $k = 7$


So the final answer is $L = 7$!

```
def count_pieces(woods, L):
    total = 0
    for w in woods:
        total += w // L
    return total

def max_wood_length(woods, k):
    if sum(woods) < k:
        return 0

    left, right = 1, max(woods)
    answer = 0

    while left <= right:
        mid = (left + right) // 2
        if count_pieces(woods, mid) >= k:
            answer = mid
            left = mid + 1
        else:
            right = mid - 1

    return answer

woods = [20, 15, 10, 17]
k = 7
print(max_wood_length(woods, k))
```

→ 7