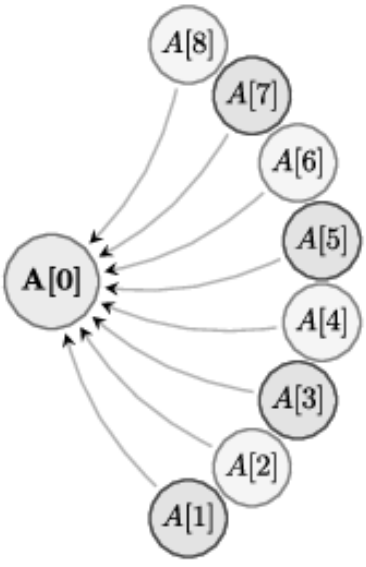
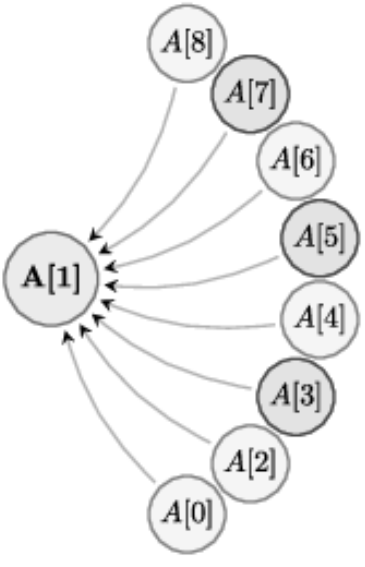
new idea

SOLUTION

Questions Game $O(n^2)$



$j = 0$



$j = 1$

$\cdots$

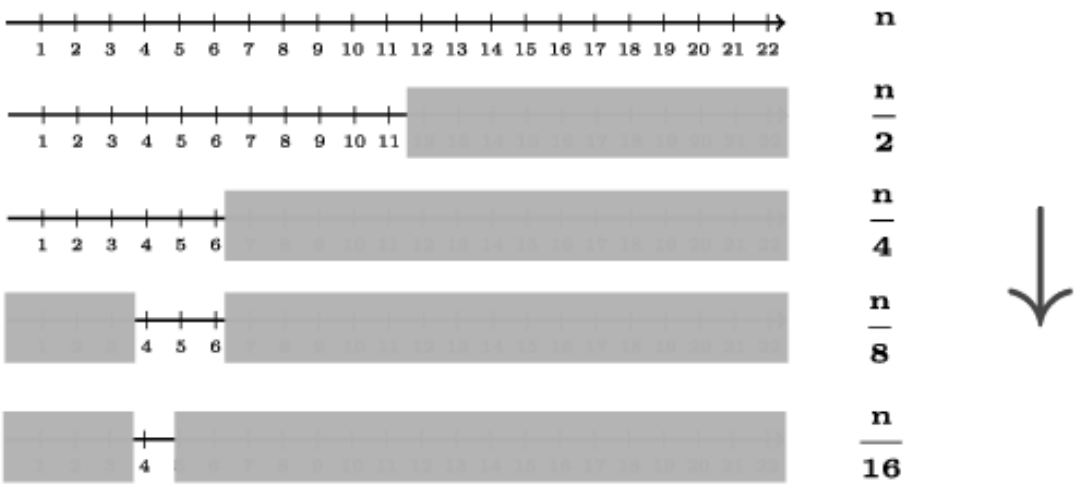
```
def computeHonestPerson_SLOW_ITERATIVE(A, isHonest):
    n = len(A)

    if n == 1: return A[0]

    for j in range(0, len(A)):
        count = 0
        for i in range(0, len(A)):
            if i != j and isHonest(A[i], A[j]):
                count += 1
        if 2*count >= len(A)-1:
            return j
```

```
def computeHonestPerson_SLOW_RECURSIVE(A, isHonest):
    n = len(A)

    if n == 1:
        return A[0]

    count = 0
    for i in range(0, n-1):
        if isHonest(A[i], A[n-1]):
            count += 1
    if 2*count >= len(A)-1:
        return A[n-1]
    else:
        return computeHonestPerson_SLOW_RECURSIVE(A[0:n-1],isHonest)
```

new idea

Divide and Conquer

Dans la recherche binaire, à chaque itération, la taille du problème est divisée par deux (passant de n à $\frac{n}{2}$).



Maintenant, nous examinons le cas où nous ne réduisons pas la **taille totale** du problème, mais où nous pouvons tout de même améliorer le temps d'exécution en utilisant diviser pour régner.

Nous devons résoudre un problème **P** sur une liste `A` de taille n.

Voici les outils à notre disposition pour trouver une solution pour problème **P** :

1. une fonction `f(B)` qui retourne la solution de **P** sur une liste `B`

Elle prend un temps $(\texttt{len(B)})^2$.

2. une fonction `g(B, solutionB, C, solutionC)` qui prend listes `B` et `C`, ainsi que leurs solutions, et retourne la solution de **P** pour la liste *combinée* `B+C`.

Elle prend un temps $\texttt{len(B)} + \texttt{len(C)}$.

Question : Quelle est la manière la plus rapide de résoudre le problème **P** sur `A`, avec $n = \texttt{len(A)}$?

Approche 0 : utiliser `f(A)` pour résoudre directement notre problème sur `A`.

temps d'exécution total : $\texttt{len(A)}^2 = n^2$

Peut-on faire mieux ? Oui !

Approche 1 : Diviser pour Régner

Étape 1: Couper `A` en deux listes `B` et `C`, de taille $n/2$ chacun

Étape 2 : Utiliser `f` pour résoudre le problème sur `B`, puis séparément, utiliser `f` pour résoudre le problème sur `C`.

temps d'exécution : $\texttt{len(B)}^2 + \texttt{len(C)}^2 = (n/2)^2 + (n/2)^2 = \mathbf{n^2/2}$

Étape 3 : Utiliser `g` pour combiner les solutions de `B` avec `C`

temps d'exécution : $\texttt{len(B)} + \texttt{len(C)} = n/2 + n/2 = \mathbf{n}$

temps d'exécution total: $\dfrac{\mathbf{n^2}}{\mathbf{2}} + \mathbf{n}$

Approche 1 est plus rapide que Approach 0 !

```
def f(A):
  ...

def g(B, solB, C, solC):
  ...


#Approach 0:
solA = f(A)

#Approach 1:
B = A[:len(A)//2]
C = A[len(A)//2:]
solB = f(B)
solC = f(C)
solA = g(B, solB, C, solC)
```

Que s'est-il vraiment passé ?

Le point essentiel est que `f` prend un temps n^2.

si n double, n^2 sera multiplié par 4 $\implies$ $(2n)^2 = 4n^2$

de même, diviser n par 2 réduira le temps d'exécution **quatre** fois $\implies$ $\left(\frac{n}{2}\right)^2 = \frac{n^2}{4}$

Alors, résoudre deux problèmes, chacun de taille $n/2$, avec `f` est deux fois plus rapide que résoudre un seule problème de taille n avec `f`

temps: $\mathbf{n^2}$ contre $2\left(\frac{n}{2}\right)^2 = \mathbf{\frac{n^2}{2}}$

Que s'est-il vraiment passé ?

Bien sûr, si nous résolvons deux problèmes séparément, nous payons un prix : nous devons utiliser `g` pour combiner leurs solutions.

Mais le point clé est que combiner les solutions ne nous coûte que n.
Au final, nous gagnons quand même !

temps d'exécution total : $\mathbf{n^2}$ contre $\mathbf{\frac{n^2}{2} + n}$

Nous améliorons donc le temps d'exécution d'un facteur presque 2.

Ce n'est pas très impressionnant . . . peut-on faire encore mieux ?

Approche 2 : Diviser pour Régner ... DEUX NIVEAUX !

Étape 1: Couper `A` en quatre listes `B, C, D, E`, chacune de taille $n/4$

Étape 2 : Utiliser `f` pour résoudre chaque `B, C, D, E` séparément

temps d'exécution : $4 \cdot (n/4)^2 = \mathbf{n^2/4}$

Étape 3 : Utiliser `g` pour fusionner `B` avec `C`. Utiliser `g` à nouveau pour fusionner `D` avec `E`

temps d'exécution : $\left(\frac{n}{4} + \frac{n}{4}\right) + \left(\frac{n}{4} + \frac{n}{4}\right) = \mathbf{n}$

Étape 4 : Utiliser `g` pour fusionner (`B+C`) avec (`D+E`).

temps d'exécution : $\frac{n}{2} + \frac{n}{2} = \mathbf{n}$

temps d'exécution total: $\frac{\mathbf{n^2}}{4} + \mathbf{2n}$

Approche 2 est plus rapide que Approche 1 !

Approche k : Diviser pour Régner ... k NIVEAUX !

Étape 1: Couper `A` en 2^k listes, chacune de taille $n/2^k$

Étape 2 : Utiliser `f` pour résoudre chaque sous-liste séparément

temps d'exécution : $2^k \cdot (n/2^k)^2 = n^2/2^k$

Étape 3 : Utiliser `g` pour fusionner deux-par-deux.

temps d'exécution : $\frac{2^k}{2}\left(\frac{n}{2^k} + \frac{n}{2^k}\right) = n$

Étape 4 : Utiliser g pour fusionner deux-par-deux.

temps d'exécution : $\frac{2^k}{4}\left(\frac{2n}{2^k}+\frac{2n}{2^k}\right)=n$

$\vdots$

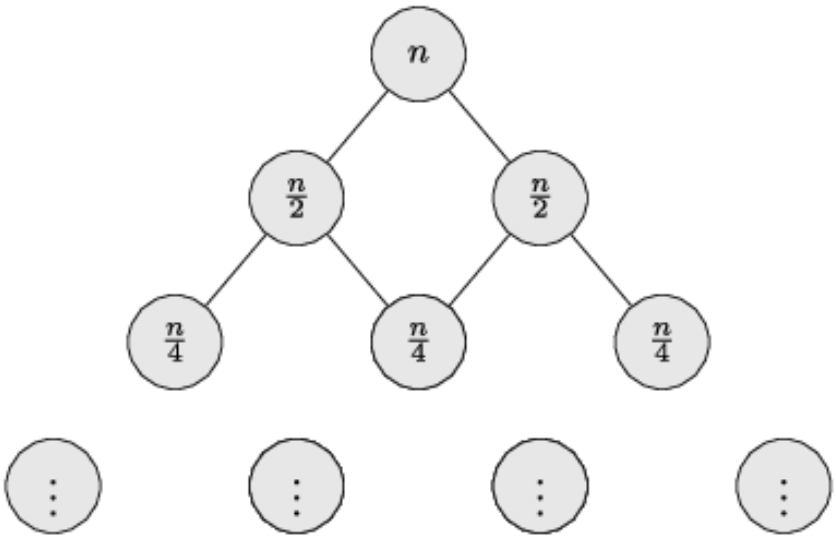
Étape finale : Utiliser g pour fusionner deux-par-deux.

temps d'exécution : $\left(\frac{n}{2}+\frac{n}{2}\right)=n$

temps d'exécution total: $\frac{n^2}{2^k}+kn$

Approche k est plus rapide que Approche k-1 !

Principe : À chaque niveau, on divise chaque sous-problème en 2 sous-problèmes de taille moitié.



Niveau 0 : 1 problème de taille n

Niveau 1 : 2 problèmes de taille $\frac{n}{2}$

Niveau 2 : 4 problèmes de taille $\frac{n}{4}$

$\vdots$

Niveau i : 2^i problèmes de taille $\frac{n}{2^i}$

il y a $\log_2 n$ niveaux

Calcul du Coût par Niveau

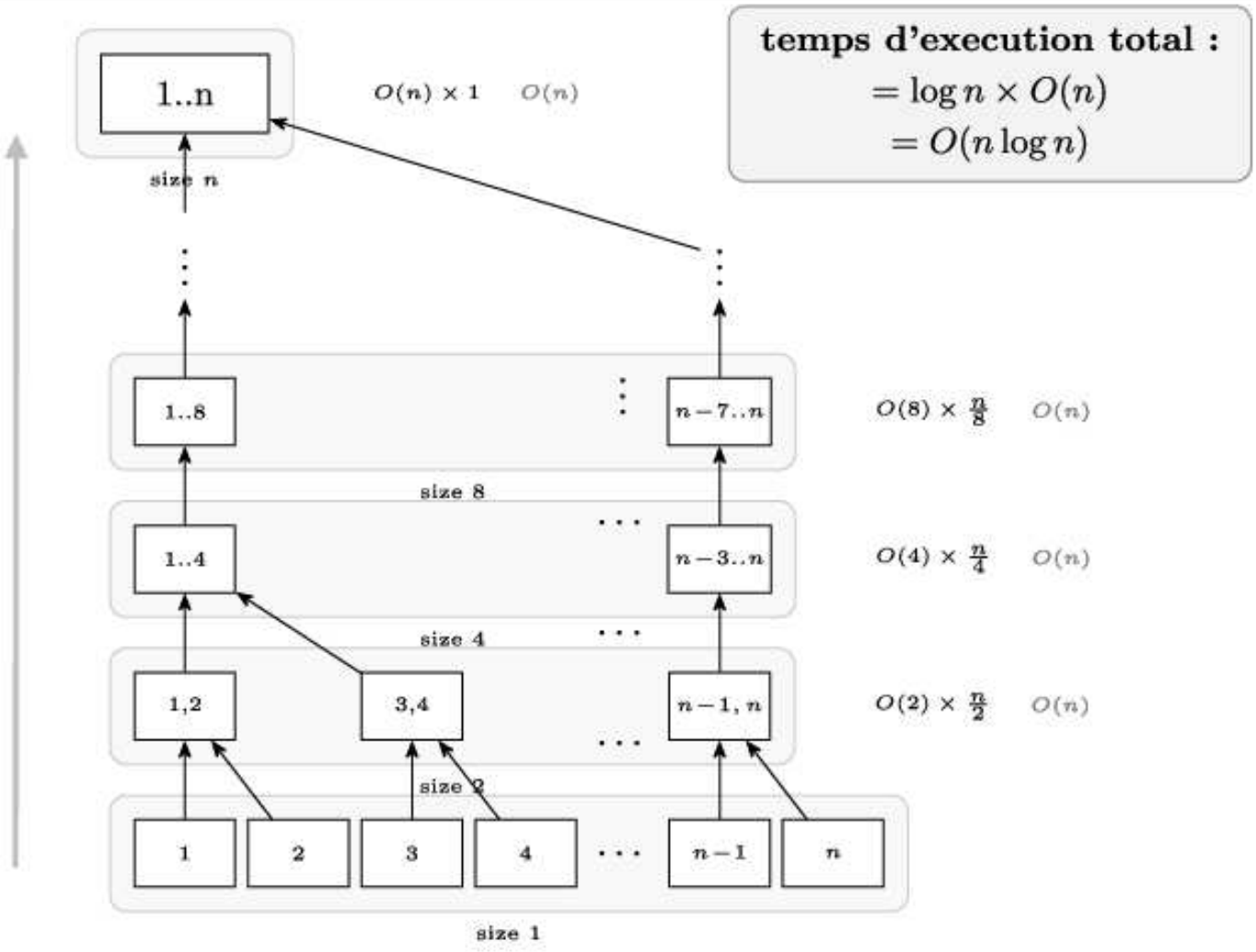
La combinaison (fusion) de deux sous-problèmes de taille $m/2$ coûte m temps.

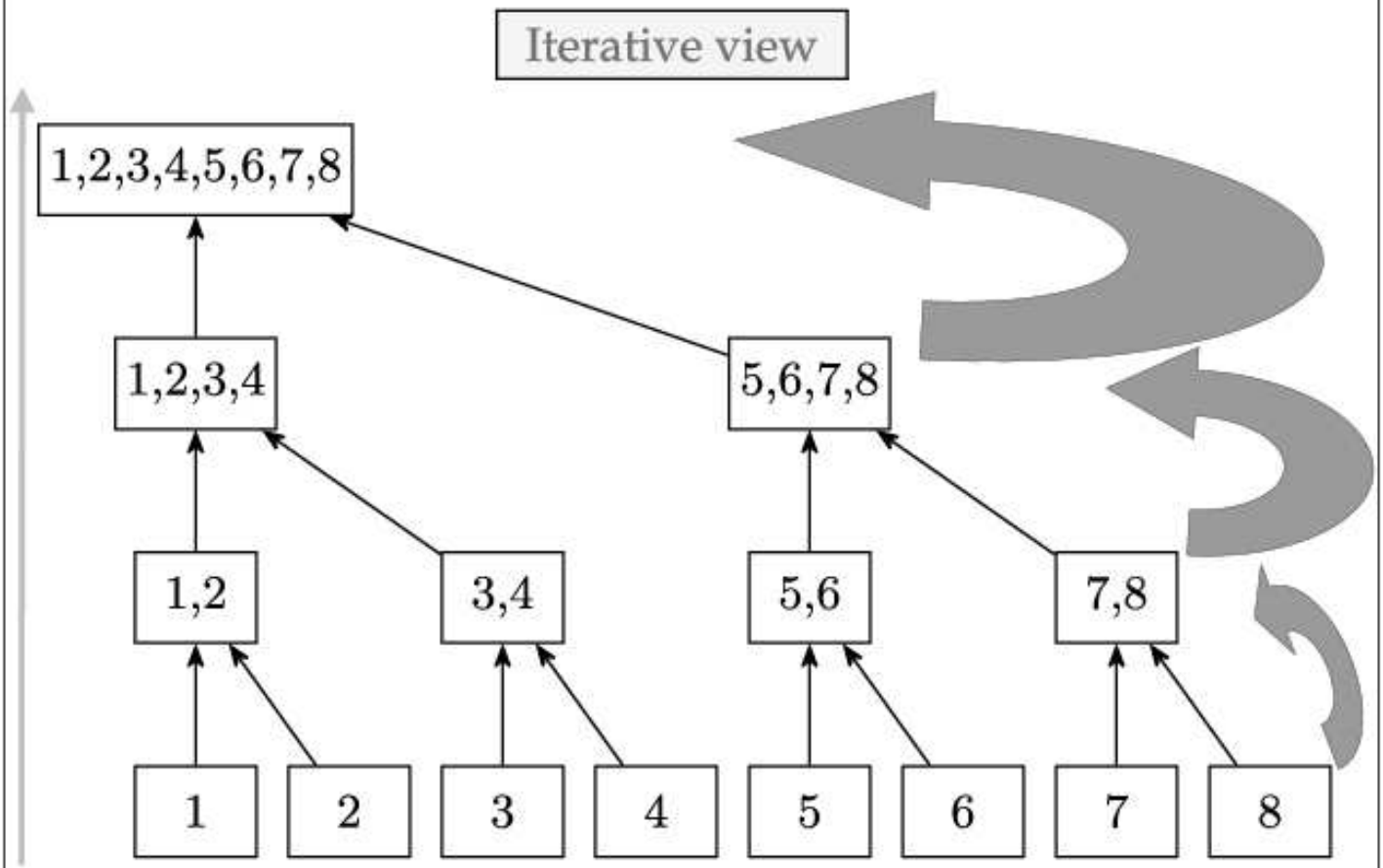
Coût au niveau i (fusion)

$= (\text{nombre de fusions}) \times (\text{coût de fusion})$

$= \frac{2^i}{2} \times \left(\frac{n}{2^i} + \frac{n}{2^i}\right)$

$= n$





Diviser pour Régner est une méthode algorithmique qui consiste à :

1. **Diviser** : Décomposer le problème en sous-problèmes plus petits, similaires au problème original.

2. **Régner** : Résoudre ces sous-problèmes récursivement. S'ils sont suffisamment petits, on les résout directement (cas de base).

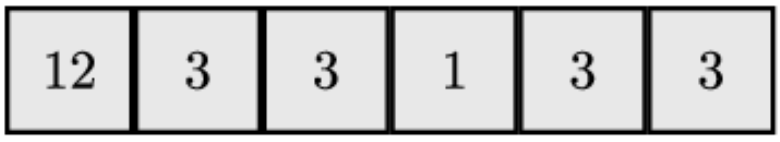
3. **Combiner** : Assembler les solutions des sous-problèmes pour obtenir la solution du problème initial.

C'est une approche **récursive** naturelle, très utilisée en algorithmique.

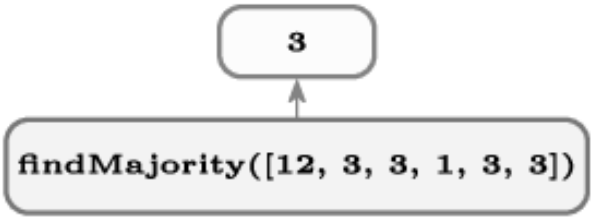
new idea

MAJORITY ELEMENT

Problème : Étant donné une liste *non-triée* `A` de `n` entiers, trouver un élément (s'il existe) qui apparaît plus de $n/2$ fois.




```
A = [12, 3, 3, 1, 3, 3]
```



Étape clé : Fusion de deux sous-listes

Soit B et C deux sous-listes de A, avec $n = |B| + |C|$.
Supposons que x est un candidat majoritaire dans B (ou `None`)
Et y est un candidat majoritaire dans C (ou `None`).

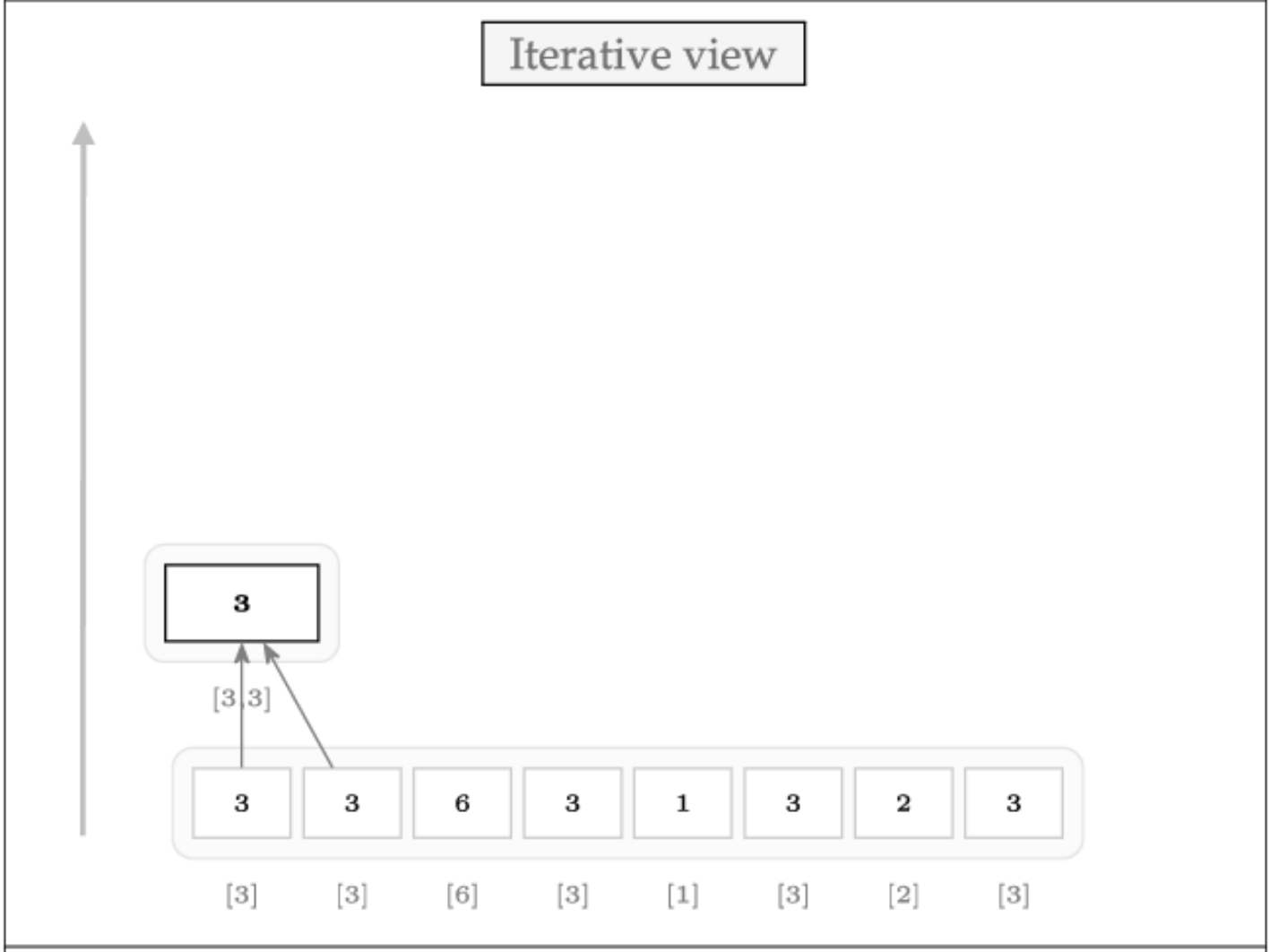
Pour déterminer le majoritaire dans $B \cup C$ (s'il existe) :

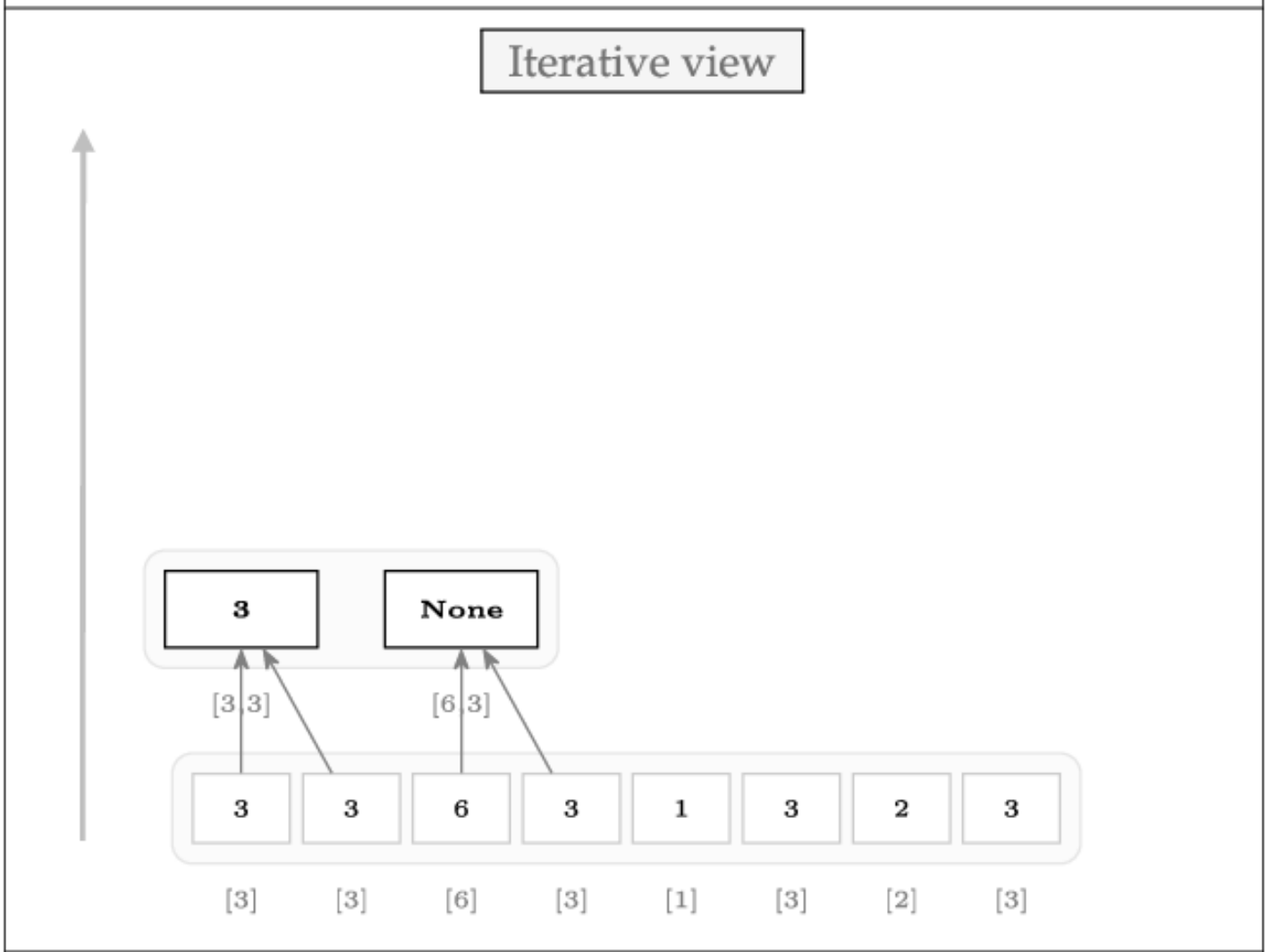
1. Compter les occurrences :
   $count1 \leftarrow$ occurrences$(x,\ B \cup C)$
   $count2 \leftarrow$ occurrences$(y,\ B \cup C)$
2. Si $count1 > n/2$: retourner x
3. Sinon si $count2 > n/2$: retourner y
4. Sinon : retourner `None` (aucun majoritaire)

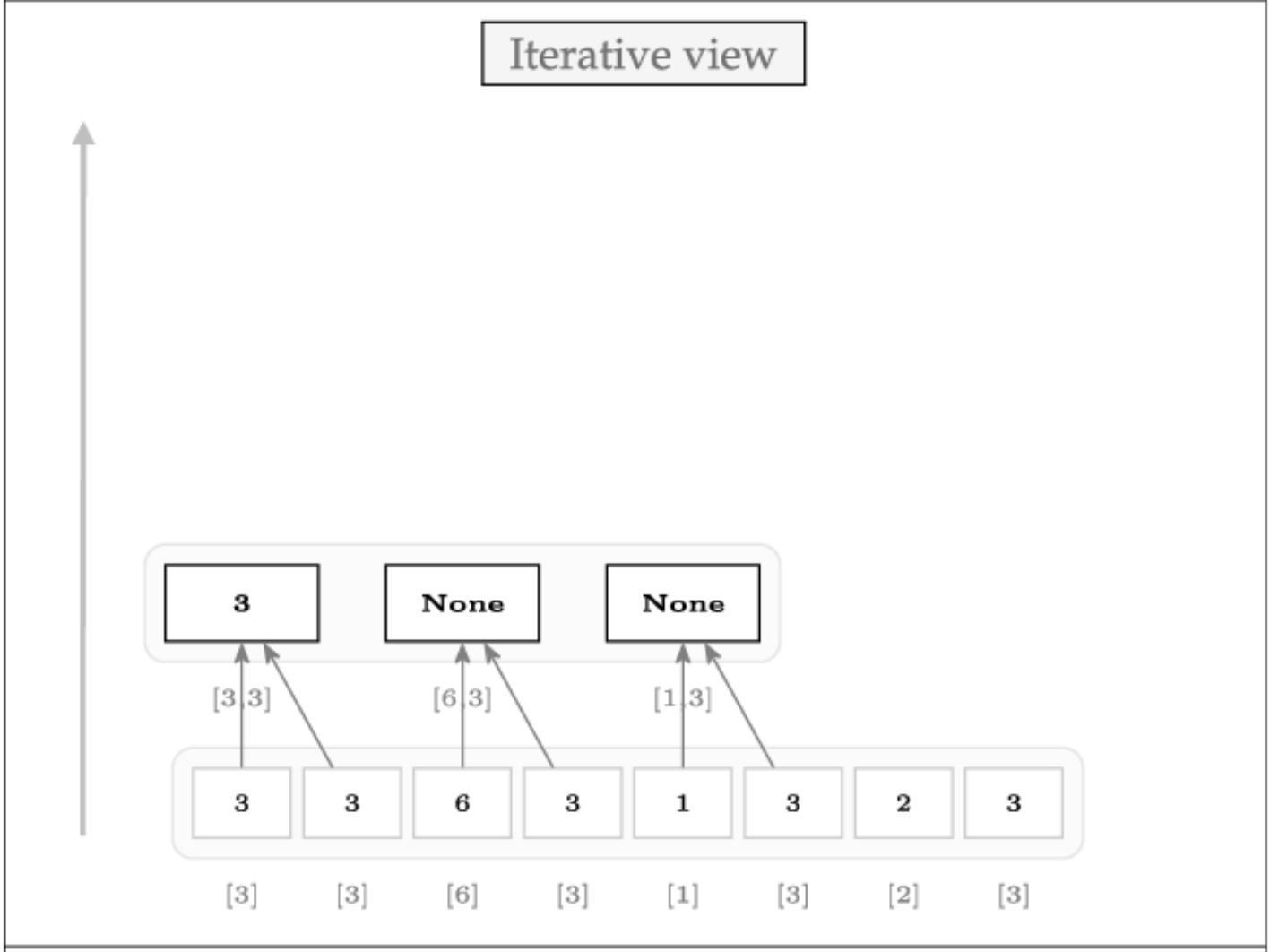
Justification : Un élément majoritaire dans $B \cup C$ doit nécessairement être majoritaire dans B ou dans C.

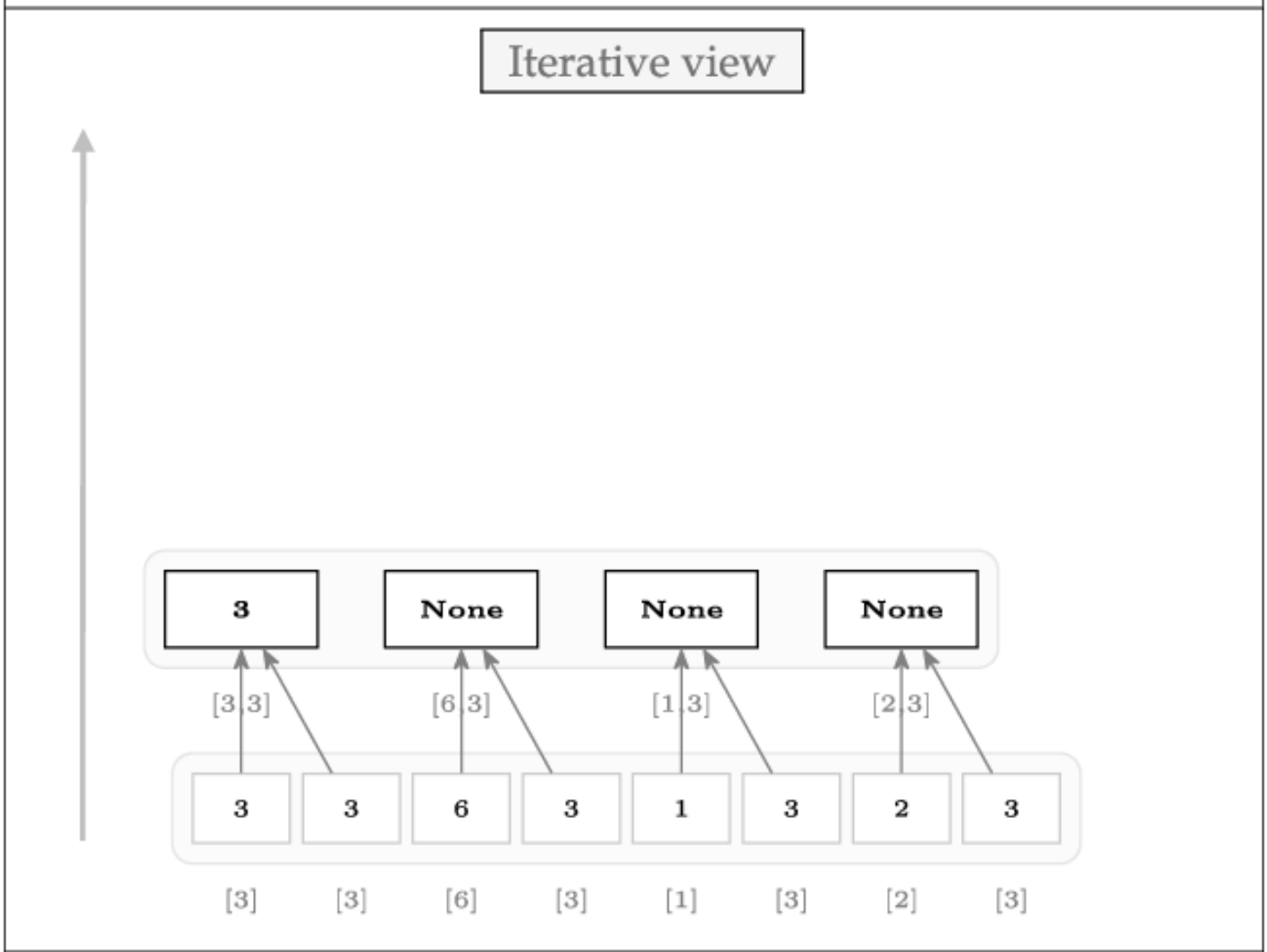
temps pour la fusion à chaque niveau : $O(n)$
nombre de niveaux : $\log_2 n$
$\Longrightarrow$ **temps d'execution total :** $O\left(n \log_2 n\right)$

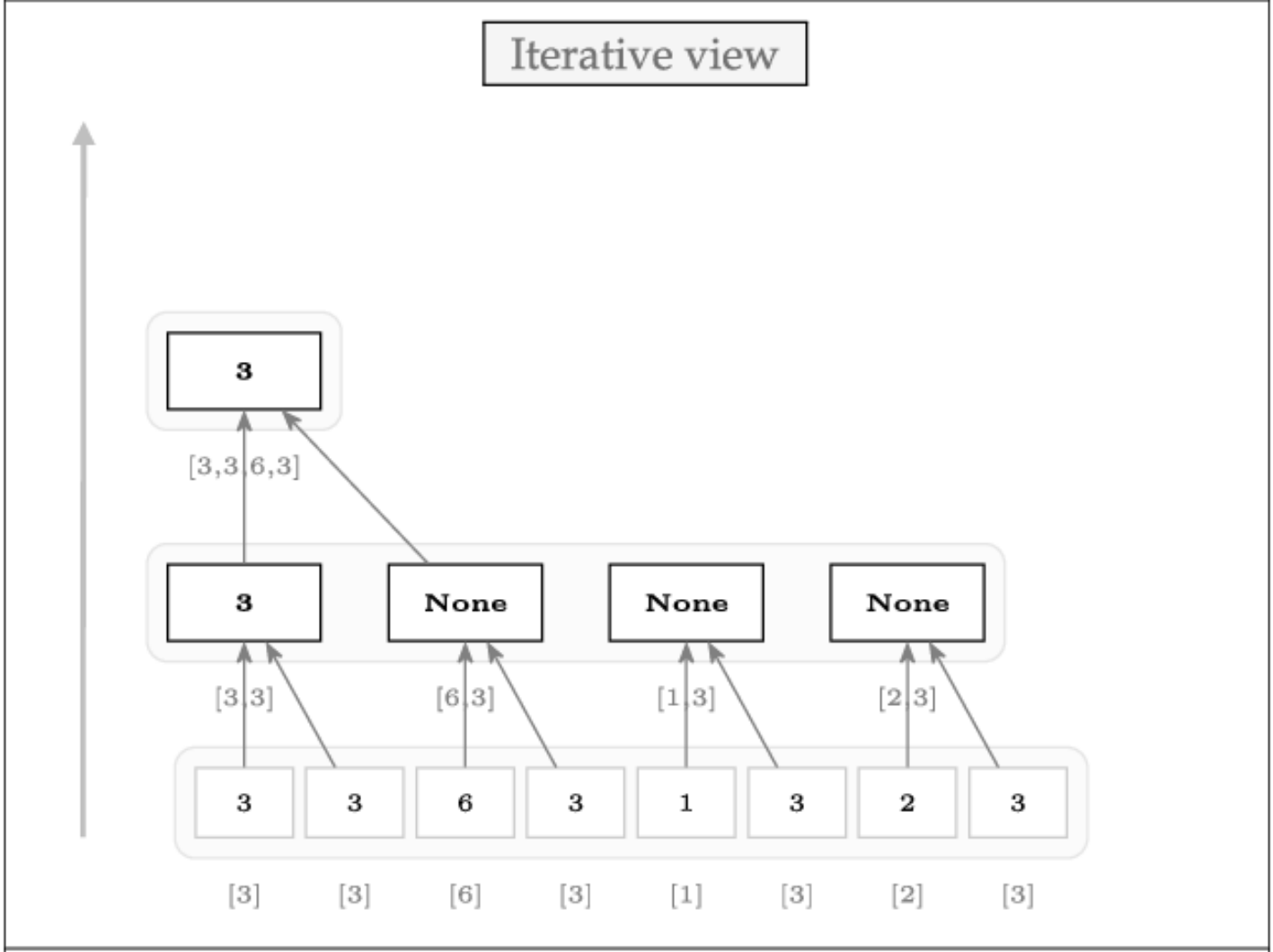
Iterative view

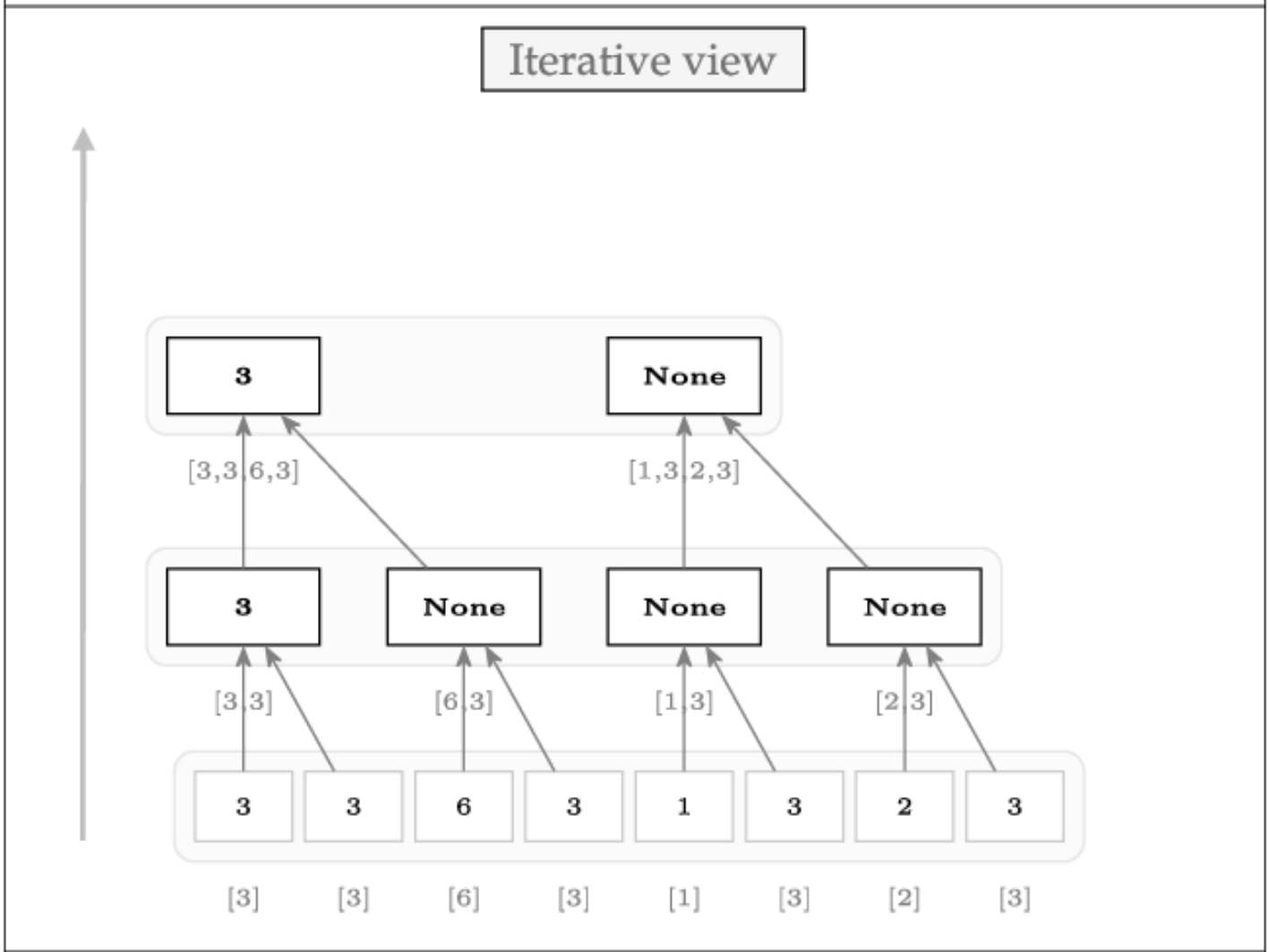

Iterative view
3
[3, 3]
3 3 6 3 1 3 2 3
[3] [3] [6] [3] [1] [3] [2] [3]

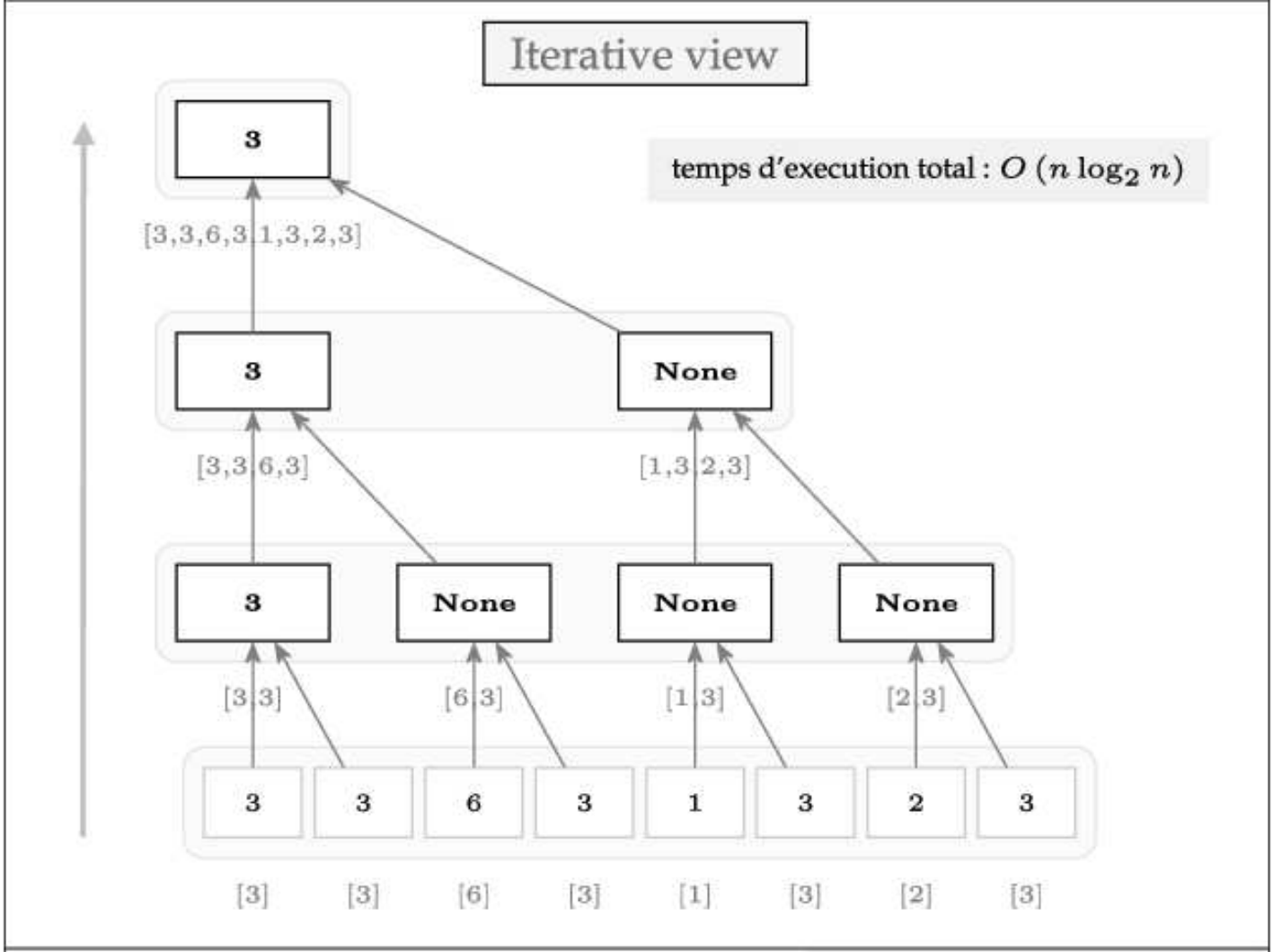

Iterative view
3
None
[3, 3]
[6, 3]
3 3 6 3 1 3 2 3
[3] [3] [6] [3] [1] [3] [2] [3]

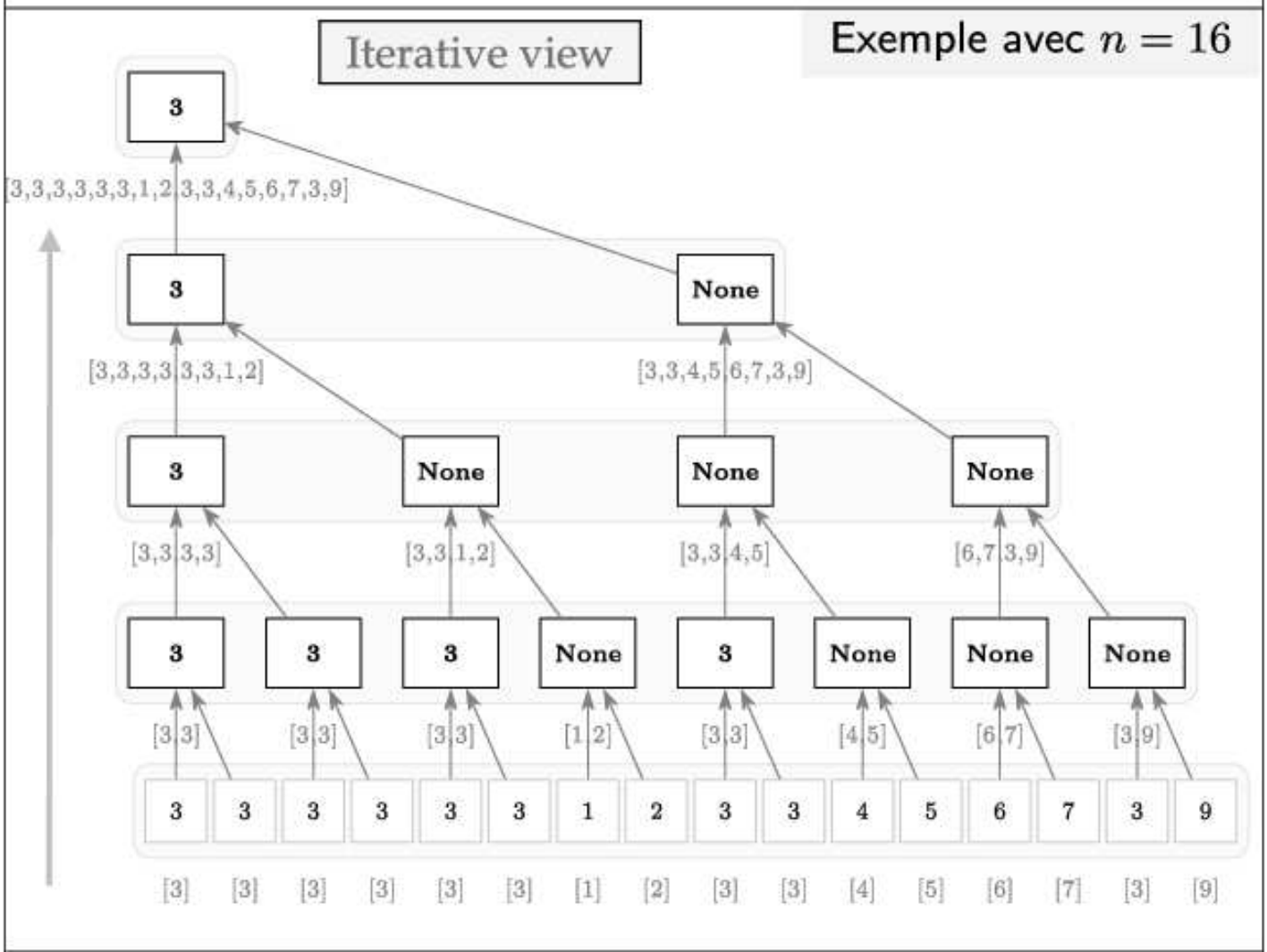

Iterative view
3
None
None
[3 3]
[6 3]
[1 3]
3
3
6
3
1
3
2
3
[3]
[3]
[6]
[3]
[1]
[3]
[2]
[3]


Iterative view
3
None
None
None
[3 3]
[6 3]
[1 3]
[2 3]
3
3
6
3
1
3
2
3
[3]
[3]
[6]
[3]
[1]
[3]
[2]
[3]


Iterative view
3
[3,3 6,3]
3
None
None
None
[3 3]
[6 3]
[1 3]
[2 3]
3
3
6
3
1
3
2
3
[3]
[3]
[6]
[3]
[1]
[3]
[2]
[3]


Iterative view
3
None
[3,3 6,3]
[1,3 2,3]
3
None
None
None
[3 3]
[6 3]
[1 3]
[2 3]
3
3
6
3
1
3
2
3
[3]
[3]
[6]
[3]
[1]
[3]
[2]
[3]


Iterative view
temps d'execution total : $O(n \log_2 n)$
3
[3,3,6,3,1,3,2,3]
3
None
[3,3,6,3]
[1,3,2,3]
3
None
None
None
[3,3]
[6,3]
[1,3]
[2,3]
3
3
6
3
1
3
2
3
[3]
[3]
[6]
[3]
[1]
[3]
[2]
[3]


Iterative view
Exemple avec $n = 16$
3
[3,3,3,3,3,3,1,2,3,3,4,5,6,7,3,9]
3
None
[3,3,3,3,3,3,1,2]
[3,3,4,5,6,7,3,9]
3
None
None
None
[3,3,3,3]
[3,3,1,2]
[3,3,4,5]
[6,7,3,9]
3
3
3
None
3
None
None
None
[3,3]
[3,3]
[3,3]
[1,2]
[3,3]
[4,5]
[6,7]
[3,9]
3
3
3
3
3
3
1
2
3
3
4
5
6
7
3
9
[3]
[3]
[3]
[3]
[3]
[3]
[1]
[2]
[3]
[3]
[4]
[5]
[6]
[7]
[3]
[9]

Iterative view

```python
def findMajority(A):
    if len(A).bit_count() != 1:
        return None

    niveau = int( math.log2(len(A)) )

    P = list(A)
    for j in range(1, niveau+1):
        m = len(A) // (2**j)
        N = [None]*m

        for i in range(0, m):
            count1, count2 = 0, 0
            for k in range( (i)*(2**j), (i+1)*(2**j) ):
                if A[k] == P[2*i]: count1 += 1
                if A[k] == P[2*i+1]: count2 += 1
            if count1 > 2**j / 2:
                N[i] = P[2*i]
            elif count2 > 2**j / 2:
                N[i] = P[2*i+1]
            else:
                N[i] = None
        print(P, N)
        P = N
    return P[0]

import math
print( findMajority( [3,3,3,3,3,3,1,2,3,3,4,5,6,7,3,9] ) )
```

Pour simplifier, on suppose que n est une puissance de 2.

← pour chaque niveau

← Soit m le nombre de sous-problèmes à ce niveau

← pour mettre la solution de prochaine niveau

← pour chaque sous-problème de ce niveau

← calculer la majorité pour le niveau suivant

Je sais — c'est beaucoup trop compliqué !

```python
def findMajority(A):
    if len(A).bit_count() != 1:
        return None

    niveau = int( math.log2(len(A)) )

    P = list(A)
    for j in range(1, niveau+1):
        m = len(A) // (2**j)
        N = [None]*m

        for i in range(0, m):
            count1, count2 = 0, 0
            for k in range( (i)*(2**j), (i+1)*(2**j) ):
                if A[k] == P[2*i]: count1 += 1
                if A[k] == P[2*i+1]: count2 += 1
            if count1 > 2**j/2:
                N[i] = P[2*i]
            elif count2 > 2**j / 2:
                N[i] = P[2*i+1]
            else:
                N[i] = None
        print(P)
        P = N
    return P[0]


import math
print( findMajority( [3,3,3,3,3,3,1,2,3,3,4,5,6,7,3,9] ) )
```

temps d'execution total : $O\left(n \log_2 n\right)$

Un cauchemar :

de faire attention à tous les indices et à la gestion des sous-problèmes !

```
[3, 3, 3, 3, 3, 3, 1, 2, 3, 3, 4, 5, 6, 7, 3, 9]
[3, 3, 3, None, 3, None, None, None]
[3, None, None, None]
[3, None]
3
```

Recursive view

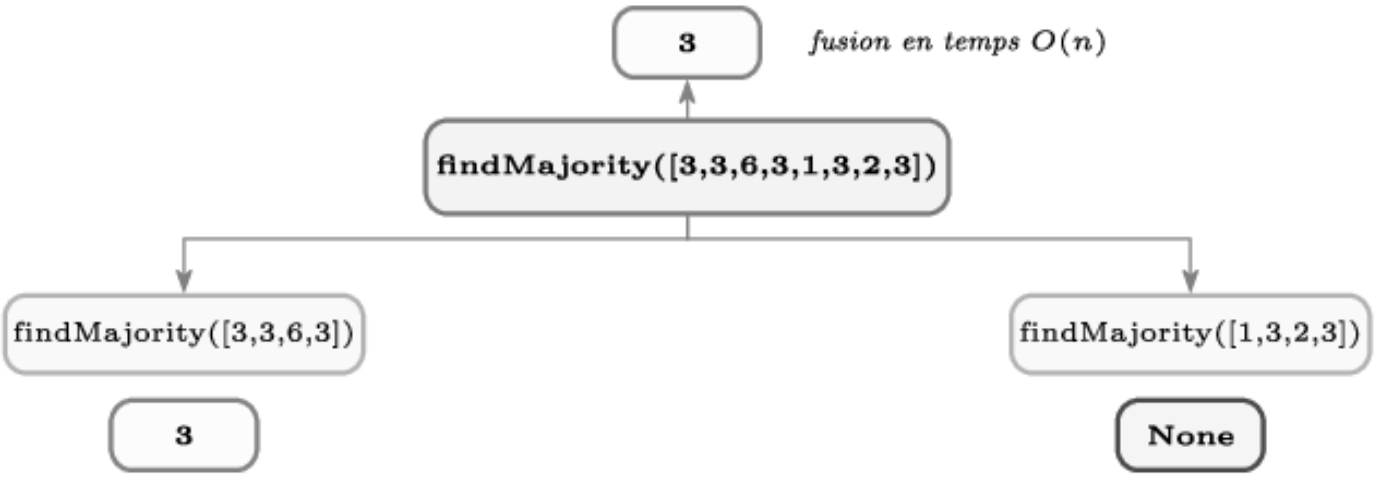
L'algorithme ascendant (bottom-up) est **compliqué à écrire** :
il faut faire attention à tous les indices et à la gestion des sous-problèmes.

Une façon équivalente : une fonction récursive !

Une fois que l'on est à l'aise avec la **récursion**, il est beaucoup plus facile de raisonner de cette manière, et le code devient bien plus simple à écrire.

Bien intégrées, les mathématiques nous aident à penser clairement !

Recursive view



temps d'execution total : $O\left(n \log_2 n\right)$

```python
def findMajority(A):
  n = len(A)

  if n == 0:  return None
  if n == 1:  return A[0]

  e1 = findMajority(A[:n//2])
  e2 = findMajority(A[n//2:])

  count1, count2 = 0, 0
  for i in range(0, n):
    if A[i] == e1: count1 += 1
    if A[i] == e2: count2 += 1

  if count1 > n/2: return e1
  if count2 > n/2: return e2
  return None

print( findMajority( [3,3,3,3,3,3,1,2,3,3,4,5,6,7,3,9] ) )
```

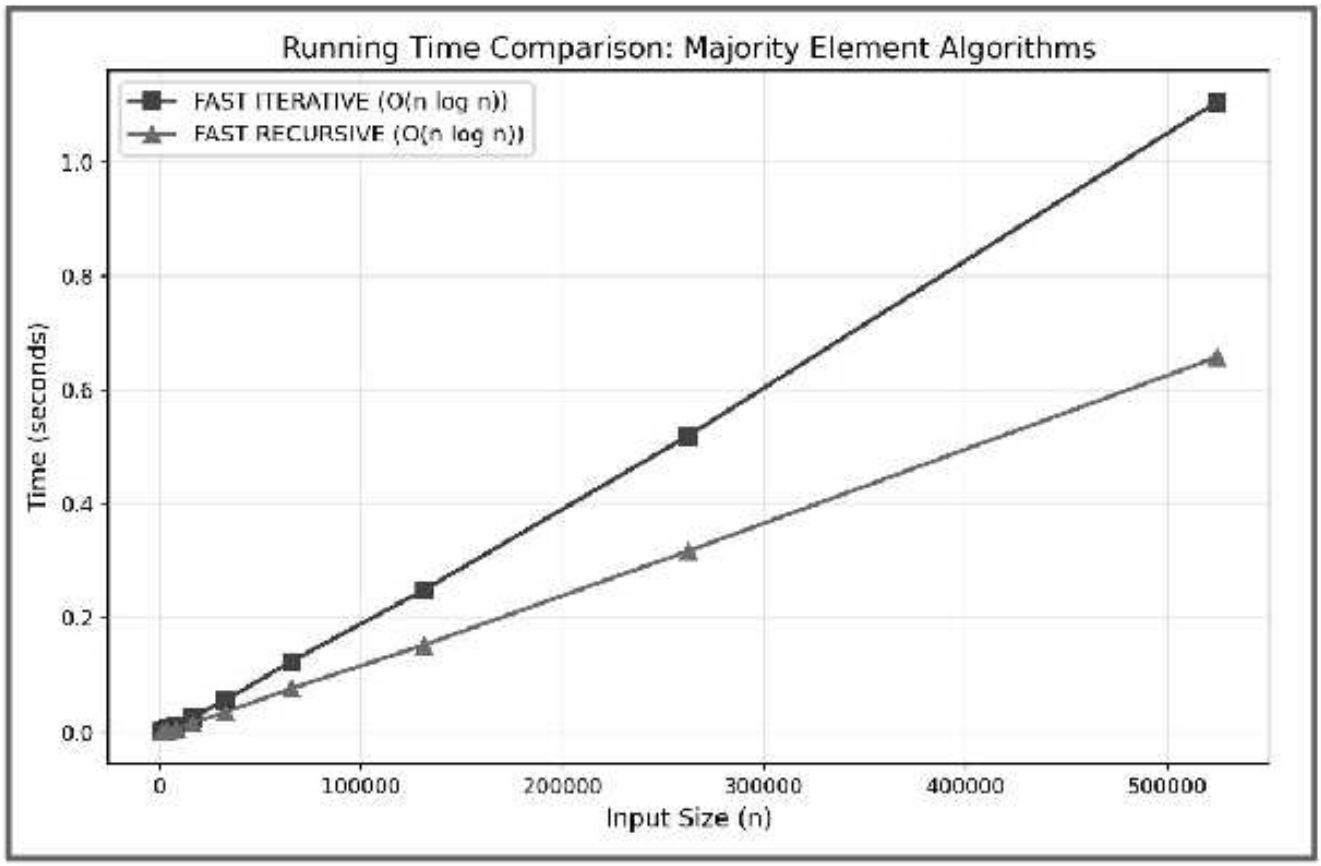
Essentiellement le même que l'algorithme itératif.

Mais beaucoup plus simple et plus beau !

3

diviser pour régner itératif : $O(n \log_2 n)$

diviser pour régner recursive : $O(n \log_2 n)$

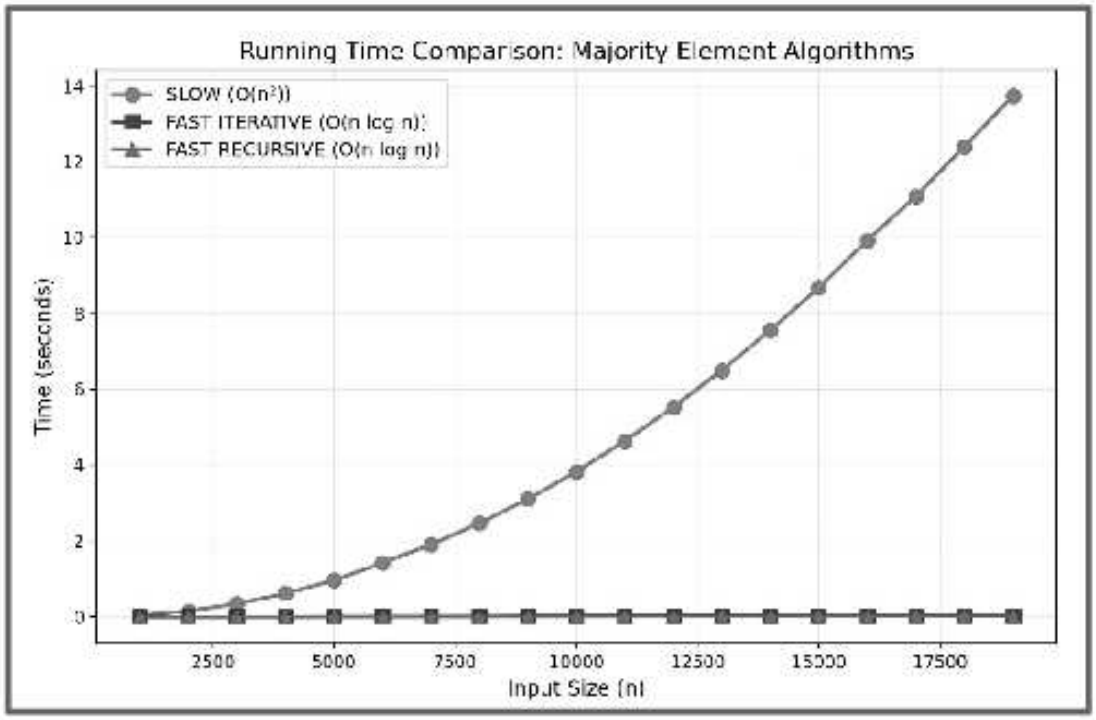


En perspective avec l'algorithme en $O(n^2)$

boucles imbriquées : $O(n^2)$

diviser pour régner itératif : $O(n \log_2 n)$

diviser pour régner recursive : $O(n \log_2 n)$



Pas pour lire, mais pour pouvoir essayer vous-meme.

```
import math, time, random, matplotlib.pyplot as plt

def findMajority_SLOW(A):
    n = len(A)
    sol = None

    for i in range(0, n):
        count = 0
        for j in range(0, n):
            if A[i] == A[j]:
                count += 1
        if count > n/2:
            sol = A[i]
    return sol

def findMajority_FAST_RECURSIVE(A):
  n = len(A)

  if n == 0: return None
  if n == 1: return A[0]

  n1 = findMajority_FAST_RECURSIVE(A[:n//2])
  n2 = findMajority_FAST_RECURSIVE(A[n//2:])

  count1, count2 = 0, 0
  for i in range(0, n):
    if A[i] == n1: count1 += 1
    if A[i] == n2: count2 += 1

  if count1 > n/2: return n1
  if count2 > n/2: return n2
  return None

def findMajority_FAST_ITERATIVE(A):
    n = len(A)
    m = len(A)
    while len(A) bit_count() != 1:
        A.append(None)

    nlevels = sum( math.log2(len(A)) )

    P = [len(A)]
    for j in range(1, nlevels+1):
        n = len(A) // (2**j)
        P = [None]*n

        for i in range(0, n):
            total, count1, count2 = 0, 0, 0
            for k in range( (i)*(2**j), (i+1)*(2**j) ):
                if k < m:
                    total += 1
                    if A[k] == P[2*i]: count1 += 1
                    if A[k] == P[2*i+1]: count2 += 1
            if count1 > total / 2:
                P[i] = P[2*i]
            elif count2 > total / 2:
                P[i] = P[2*i+1]
            else:
                P[i] = None
        P = P
    return P[0]

N, M, FI, FR = [], [], [], []
for n in range(1000, 20000, 1000):
    A = [ random.choice( [2,3] ) for j in range(0, n) ]
    N.append(n)

    print( n, _ n)
    s = time.time()
    print(findMajority_SLOW( A ))
    M.append(time.time()-s)

    s = time.time()
    print(findMajority_FAST_RECURSIVE( A ))
    FR.append(time.time()-s)

    s = time.time()
    print(findMajority_FAST_ITERATIVE( A ))
    FI.append(time.time()-s)

print( \n  +  + '#')
print(f'{'n':>10} {'SLOW (s)':>12} {'FAST ITER (s)':>14} {'FAST REC (s)':>14} )
print( + '#')
for n, m, fi, fr in zip(N, M, FI, FR):
    print(f {n:>10} {m:>12.6f} {fi:>14.6f} {fr:>14.6f} )
print( + '#')

plt.figure(figsize=(10, 6))
plt.plot(N, M, 'o-', label='SLOW O(n ²)', linewidth=2, markersize=8, color='red')
plt.plot(N, FI, 's-', label='FAST ITERATIVE O(n log n)', linewidth=2, markersize=8, color='blue')
plt.plot(N, FR, '^-', label='FAST RECURSIVE O(n log n)', linewidth=2, markersize=8, color='green')
plt.xlabel('Input Size (n)', fontsize=12)
plt.ylabel('Time (seconds)', fontsize=12)
plt.title('Running Time Comparison: Majority Element Algorithms', fontsize=14)
plt.legend(fontsize=11)
plt.grid(True, alpha=0.3)
plt.show()
```

new idea

MERGE SORT

Recursive view

```
def merge(A, B):
    C = []
    a, b = 0, 0
    for i in range(0, len(A)+len(B)):
        if (b == len(B)) or (a < len(A) and A[a] <= B[b]):
            C.append(A[a])
            a += 1
        elif (a == len(A)) or (b < len(B) and B[b] <= A[a]):
            C.append(B[b])
            b += 1
    return C

def mergeSort_RECURSIVE(A):
    if len(A) <= 1:
        return A

    mid = len(A) // 2
    L = mergeSort_RECURSIVE(A[:mid])
    R = mergeSort_RECURSIVE(A[mid:])

    return merge(L, R)


print( mergeSort_RECURSIVE([65,1,44,32,55] ))
```

Iterative view

```
def merge(A, B):
    C = []
    a, b = 0, 0
    for i in range(0, len(A)+len(B)):
        if (b == len(B)) or (a < len(A) and A[a] <= B[b]):
            C.append(A[a])
            a += 1
        elif (a == len(A)) or (b < len(B) and B[b] <= A[a]):
            C.append(B[b])
            b += 1
    return C

def mergeSort_ITERATIVE(A):
    s = 1
    while s < len(A):
        i = 0
        while (i < len(A)):
            A[i:i+2*s] = merge(A[i:i+s], A[i+s:i+2*s])
            i += 2*s
        s = 2*s

B = [9, 1, 53, 81, 16, 51, 12, 9]
mergeSort_ITERATIVE(B)
print(B)
```