INF1 : Algorithmique et Programmation

Cours 11 :
Algorithmes de tri

Domitile Lourdeaux

Université de technologie de Compiègne

Printemps 2024

Motivations

Trier un tableau d'éléments de même type

- Le type des éléments doit être muni d'une relation d'ordre

Exempble

- Avant : 8 12 5 35 21 3
- Après : 3 5 8 12 21 35

Algorithme (s) ?

Remarque

Dans la suite de ce cours, les exemples seront ssouvent présentés à l'aide de tableaux d'entiers ou de chaines de caractères. Les algorithmes peuvent cependant être appliqués à tout type de données muni d'une relation d'ordre

Principe

- On suppose qu'on connaît le nombre d'éléments n
- On cherche le minimum
- On le place en premier élément (on échange)
- On cherche le minimum suivant
- On le place en 2ème position
- Et ainsi de suite

Simulation

8	12	5	35	21	3
3	12	5	35	21	8
3	5	12	35	21	8
3	5	8	35	21	12
3	5	8	12	21	35
3	5	8	12	21	35

A voir : `https://interstices.info/les-algorithmes-de-tri/`

Algorithme (1)

Algorithm 1 Tri par sélection

n = longeur du tableau
pour i allant de 1 à $n-1$ (exclu) faire
    // Chercher l'indice du minimum entre l'indice i et la fin du tableau
1   ...

    // Permuter l'élément situé à l'indice i avec le minimum
2   ...

Algorithme (2)

Algorithm 2 Tri par sélection

```
n = longueur(tab)
pour i allant de 1 à n − 1 (exclu) faire
    // Chercher l'indice du minimum entre l'indice i et la
       fin du tableau
3   imin ← i
4   pour j allant de i + 1 à n (exclu) faire
5       si tab[j] < tab[imin] alors
6           imin ← j
    // Permuter l'élément situé à l'indice i avec le minimum
7   temp = tab[i]
8   tab[i] = tab[imin]
9   tab[imin] = temp
```

Version Python

A faire en TD...

Complexité

Temps de calcul du tri par sélection

- En fonction du nombre n d'éléments à trier

Il faut comparer : `t[`i`]` à `t[`$i+1$`]`, `t[`$n+2$`]` ... `t[n]`

- soit $n-1$ comparaisons à chaque passage dans la boucle interne
- donc $\Sigma_1^{n-1}(n-i)$ soit $\frac{n*(n-1)}{2}$

La complexité est donc en $\mathcal{O}(n^2)$ à la fois dans le meilleur des cas, en moyenne et dans le pire des cas

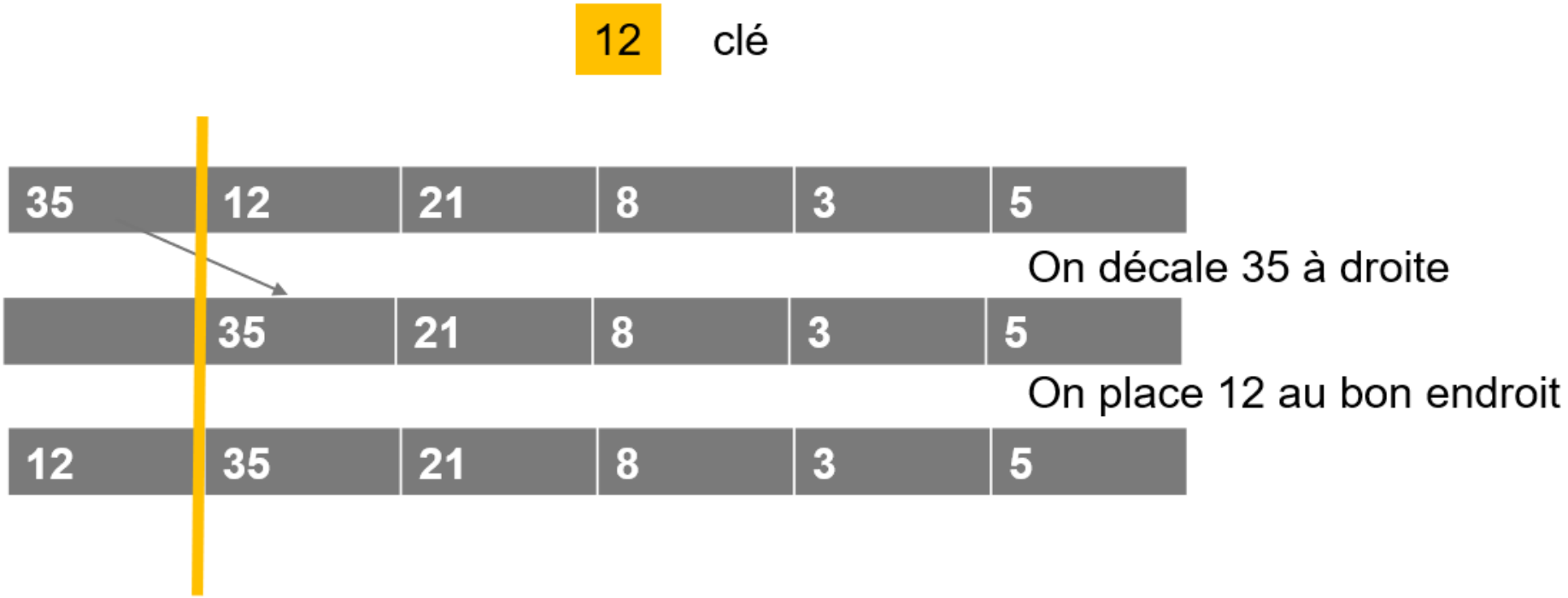
Principe

- On considère que les éléments avant i sont ordonnés (seul un élément est ordonné au début)
- On regarde l'élément i et s'il est plus petit que l'élément $i-1$ on l'insère au bon endroit dans les éléments ordonnés

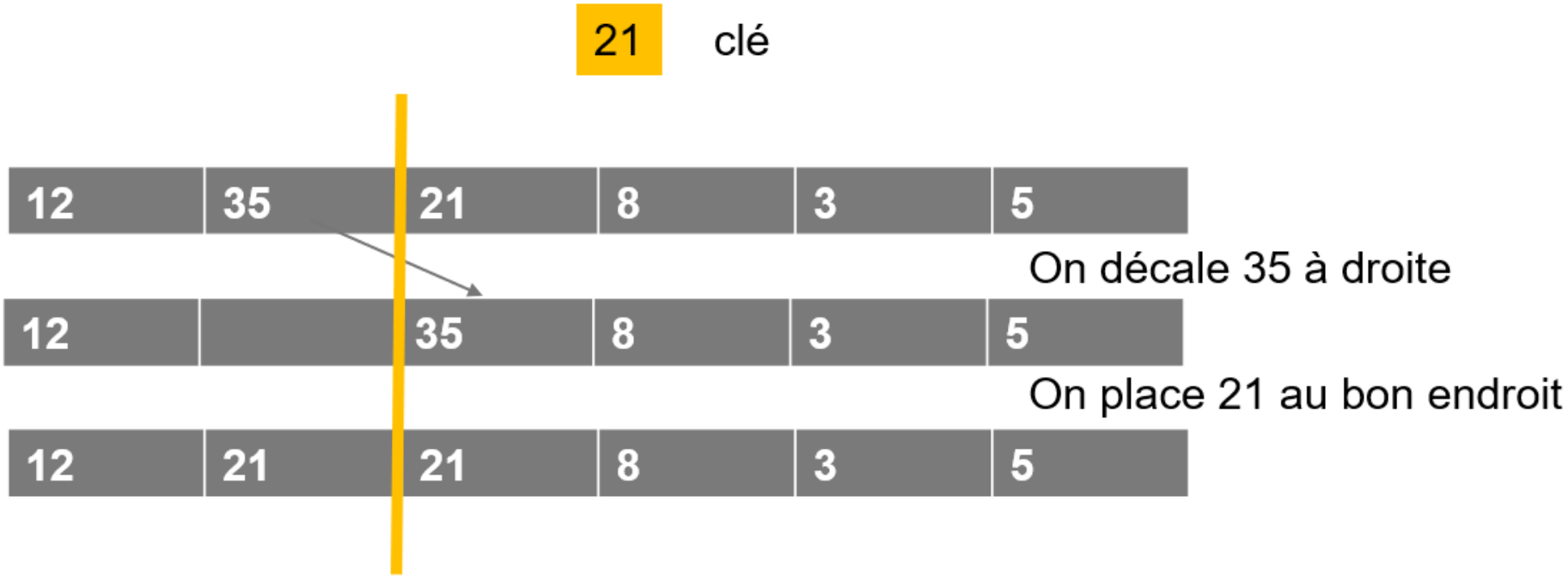
A voir : `https://interstices.info/les-algorithmes-de-tri/` Ou :

`https://youtu.be/bRPHvWgc6YM`

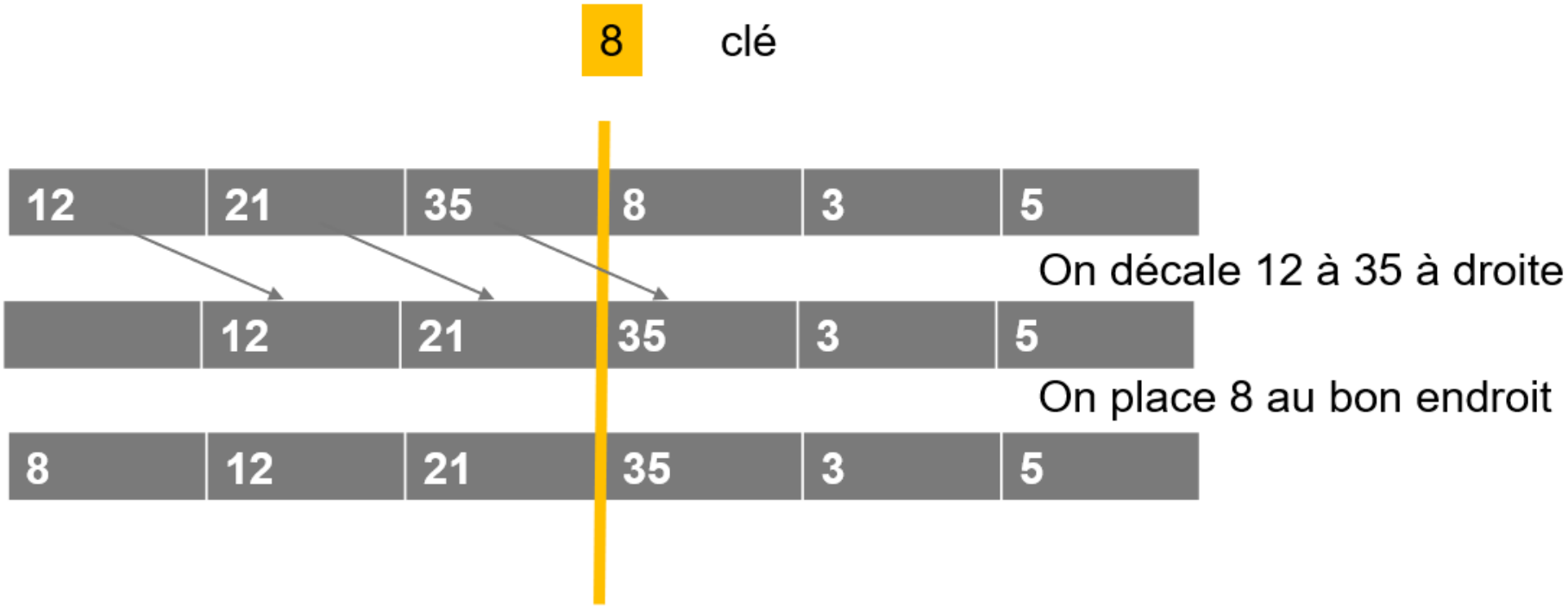
Simulation (1)



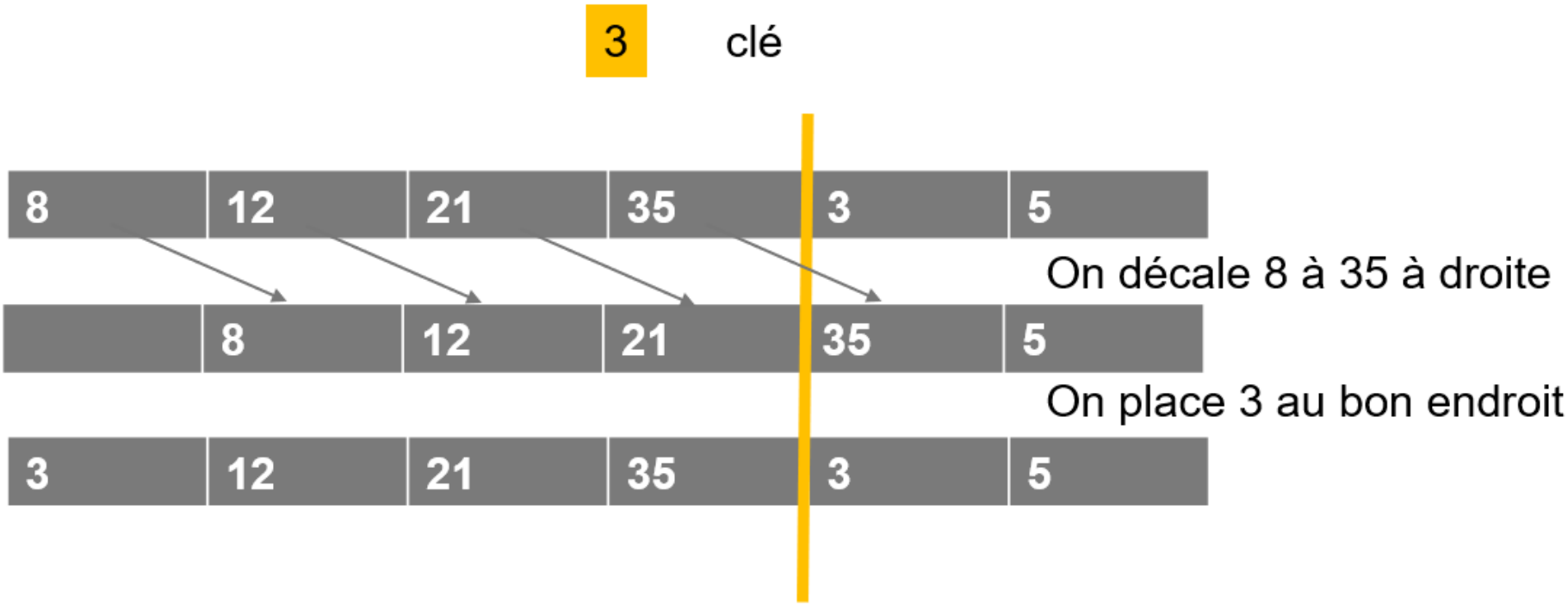
Simulation (2)



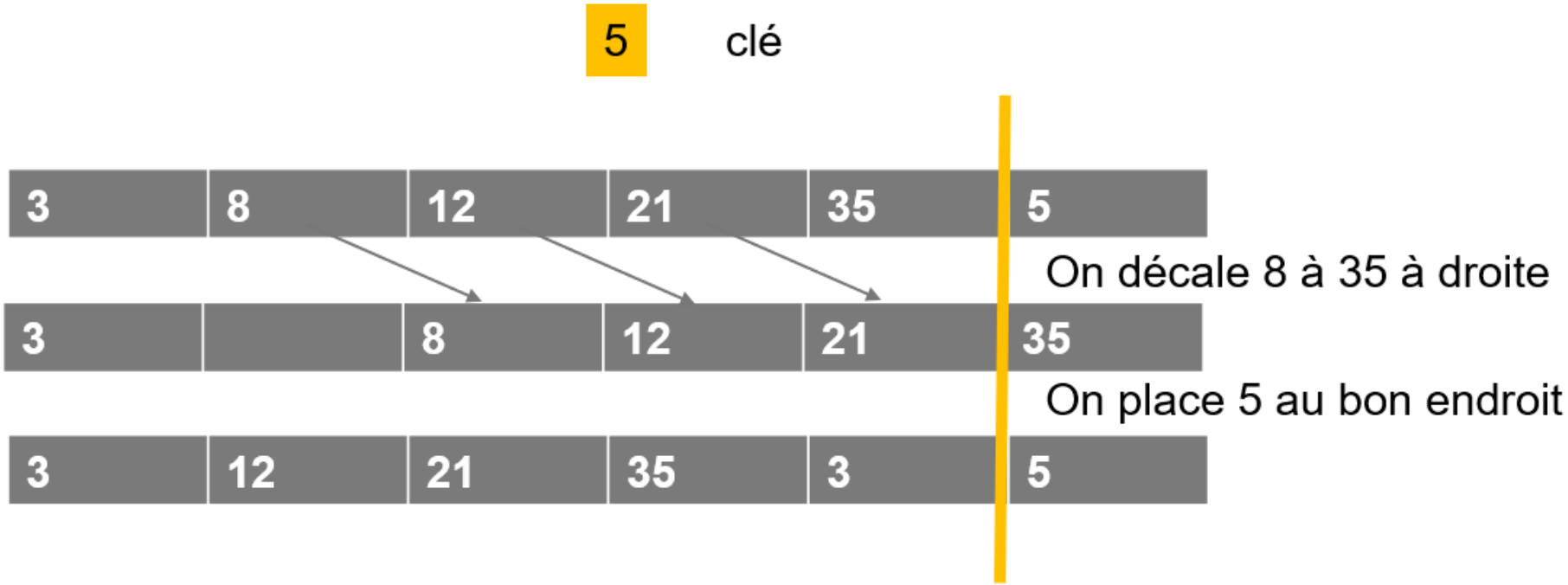
Simulation (3)



Simulation (4)



Simulation (5)



Algorithme (1)

Algorithm 3 Tri par insertion

n = longueur du tableau

```
pour i allant de 1 à n (exclu) faire
    // On considère que les éléments avant i sont ordonnés
    // clé = élément i
    // Décaler tous les éléments plus grands que la clé vers
       la droite
    ...

    // Placer la clé au bon endroit
```

Algorithme (2)

Algorithm 4 Tri par insertion

```
n = longueur(t)
pour i allant de 1 à n (exclu) faire
    // On considère que les éléments avant i sont ordonnés
11  cle ← t[i]
12  k = i − 1
13  tant que k >= 0 & cle < t[k] faire
14      t[k + 1] = t[k]
15      k− = 1
16  t[k + 1] = cle
```

Version Python

A faire en TD...

Listes d'autres types éléments

```
t = [35, 12, 21, 8, 3, 5]

t = ["Yasmine", "Armand", "Mathieu", "Clara", "Mel", "Runzhao"]

peintres = [{"nom" : "Da Vinci", "naissance":1452, "mort":1519},
            {"nom" : "Michelangelo", "naissance":1475, "mort":1564},
            {"nom" : "Raphaël", "naissance":1483, "mort":1520},
            {"nom" : "Bellini", "naissance":1430, "mort":1516}]
```

Algorithme - tri de dictionnaires

Algorithm 5 Tri de dictionnaires par insertion

```
n = longueur(t)
pour i allant de 1 à n (exclu) faire
    // On considère que les éléments avant i sont ordonnés
17  cle ← t[i]
18  k = i − 1
19  tant que k >= 0 & cle[cledico] < t[k][cledico] faire
20      t[k + 1] = t[k]
21      k− = 1
22  t[k + 1] = cle
```

Complexité algorithme de tri par insertion

Le nombre d'échanges dépend plus fortement de l'ordre initial des éléments dans le tableau :

- Meilleur des cas : une comparaison à chaque insertion, donc $n-1$ comparaisons
- Moyenne : $n-1+\frac{n*(n-1)}{4}$
- Pire des cas : $\frac{n*(n-1)}{2}$

La complexité reste en $\mathcal{O}(n^2)$

Principe

Principe

- On échange les éléments 2 à 2 en les réordonnant
- Les éléments mal classés remontent dans la liste comme des bulles à la surface d'un liquide

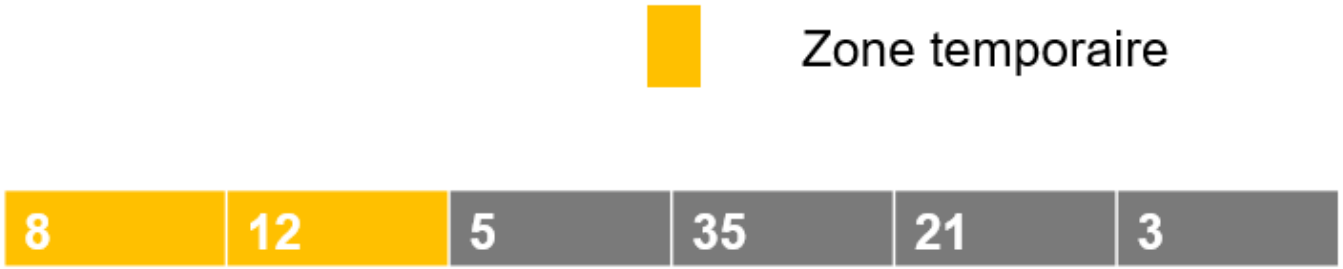
Efficacité

- Dépend du tableau initial
- Efficace si le tableau est presque trié

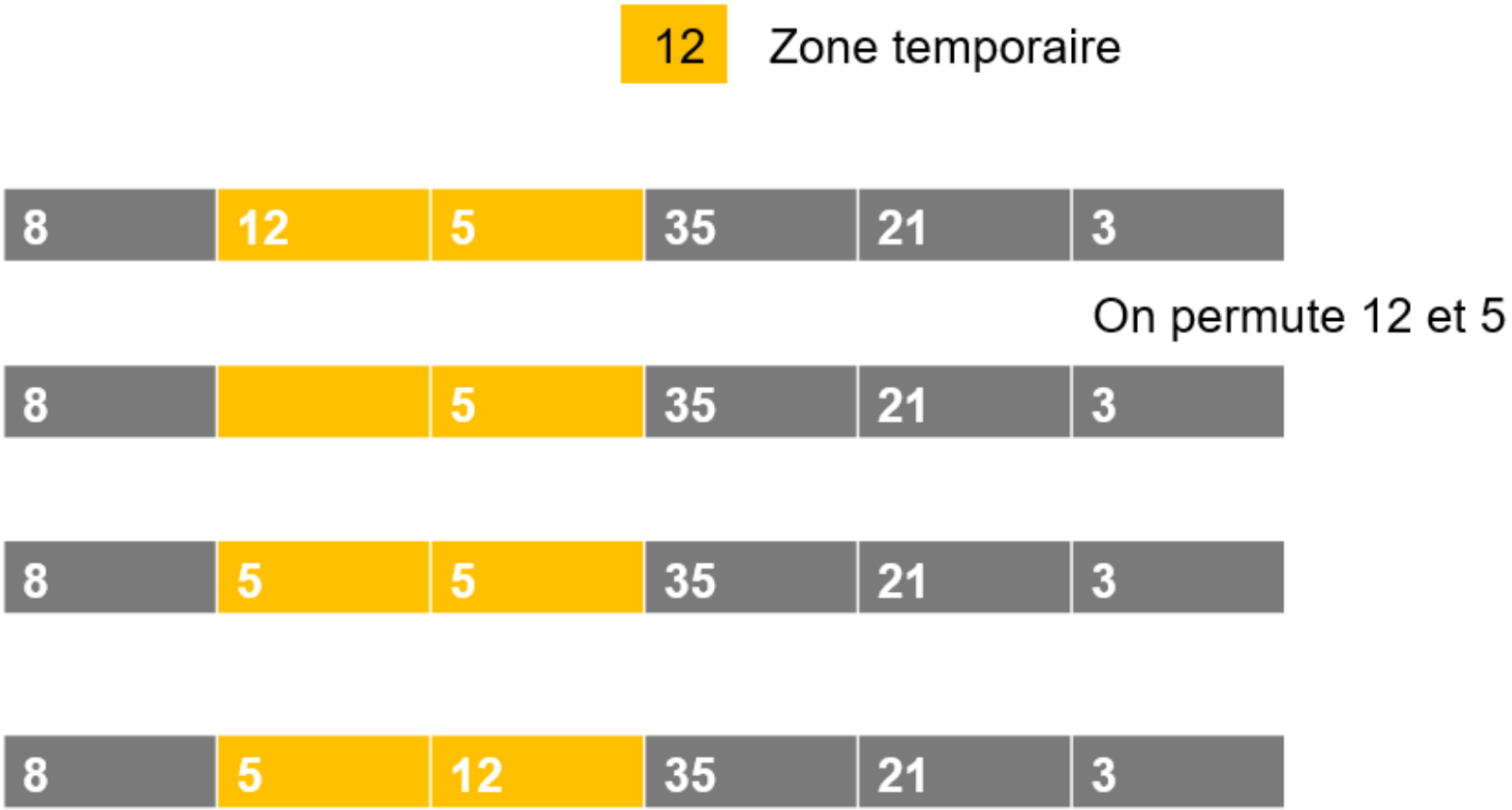
A voir :

- `https://interstices.info/les-algorithmes-de-tri/`
- `https://youtu.be/48KD2VnKeg4`
- `http://lwh.free.fr/pages/algo/tri/tri_bulle.html`
- `https://lwh-21.github.io/Algos/algorithmes%20de%20tri/bubblesort_fr.html`

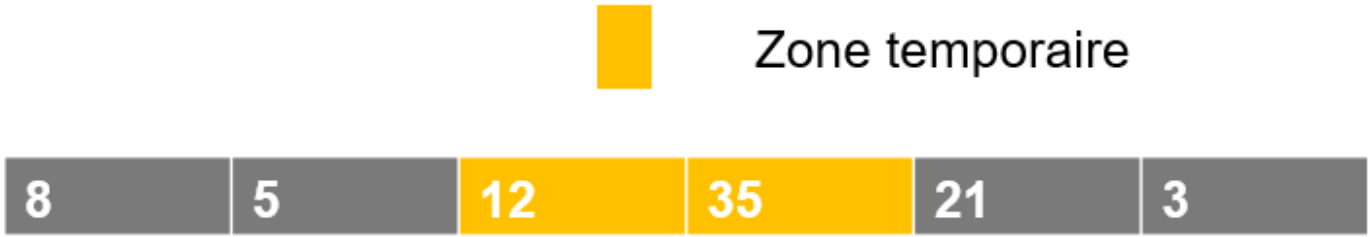
Simulation (1)



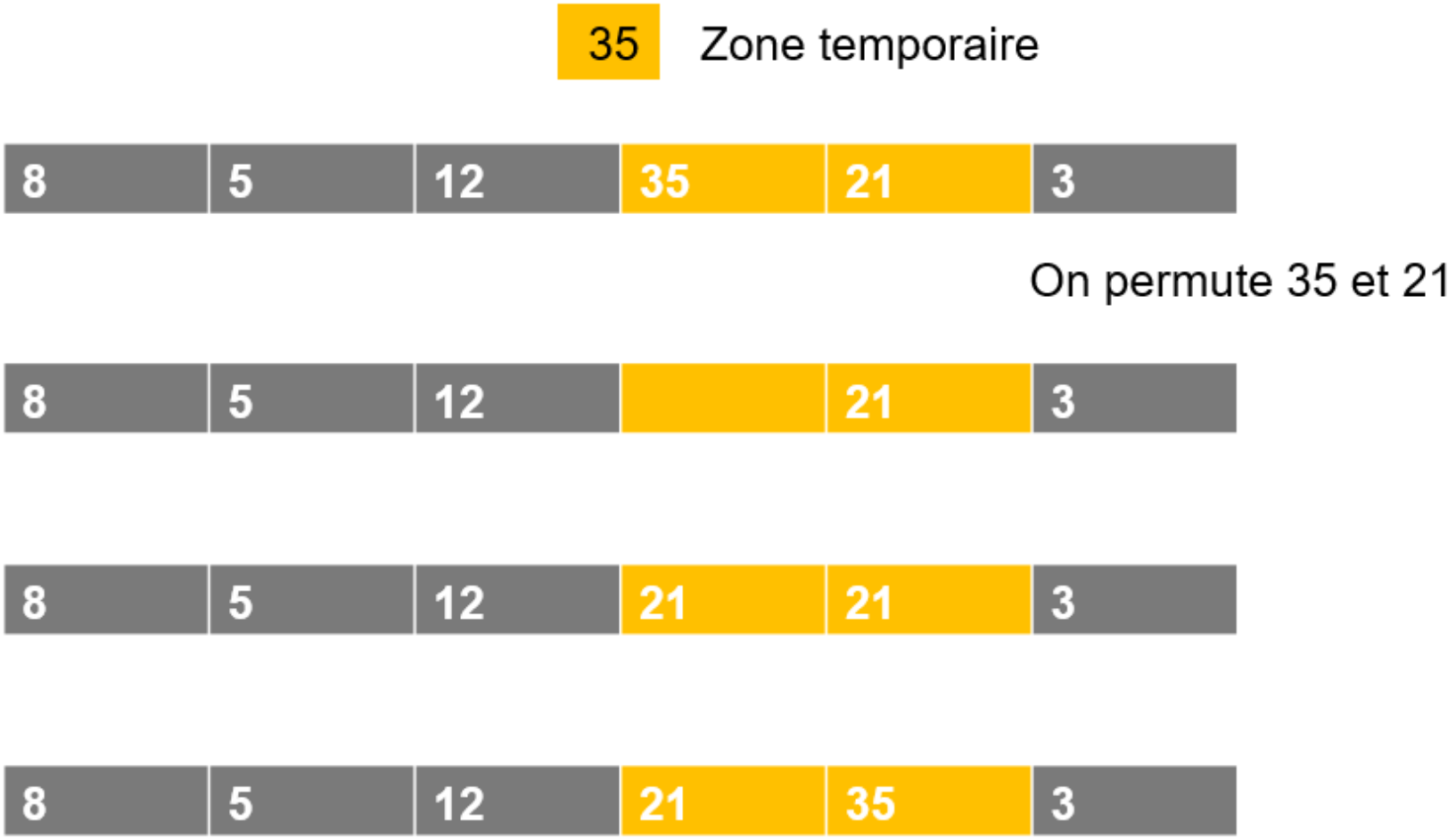
Simulation (2)



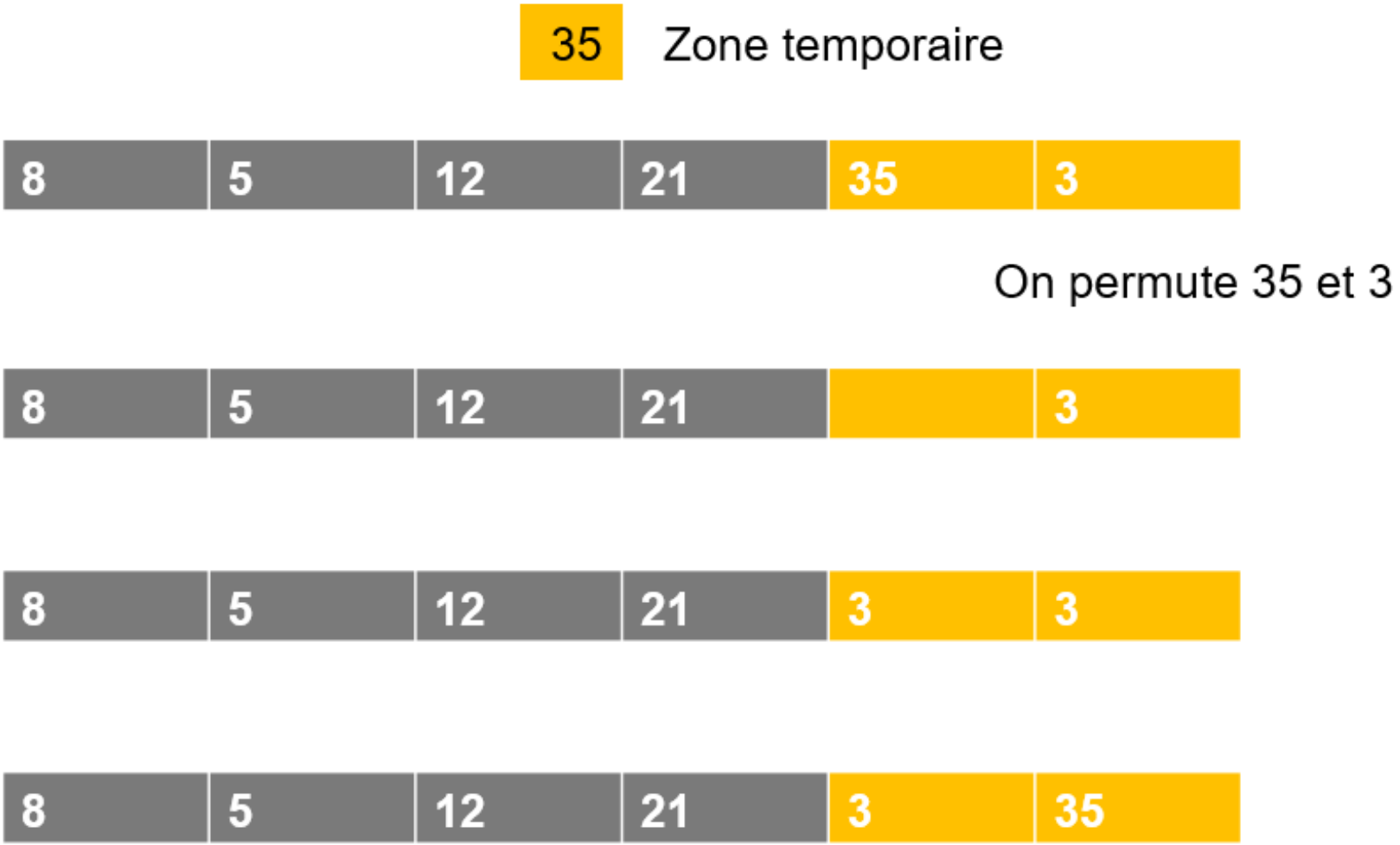
Simulation (3)



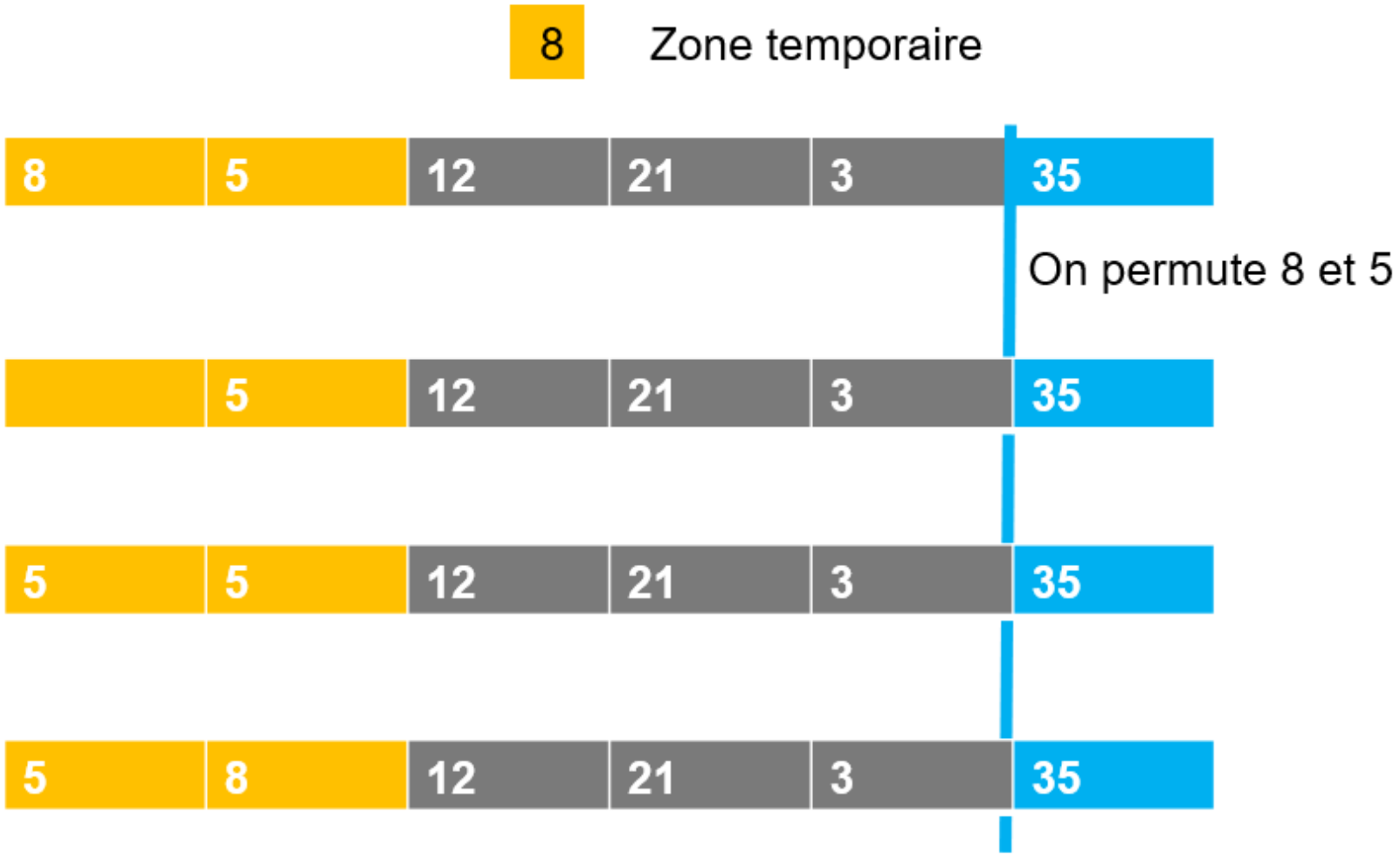
Simulation (4)



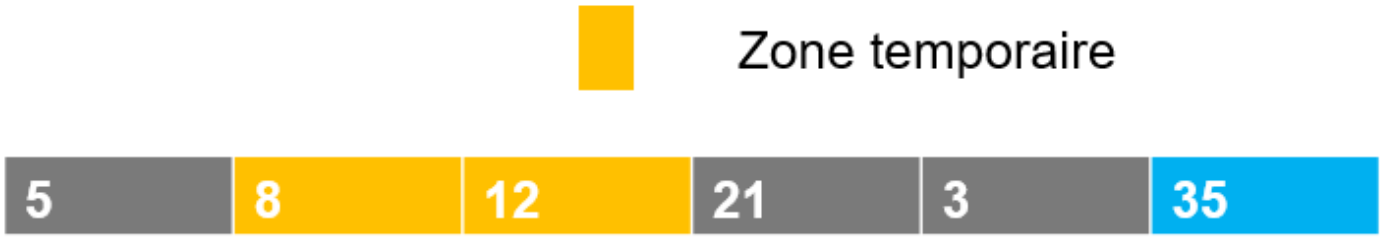
Simulation (5)



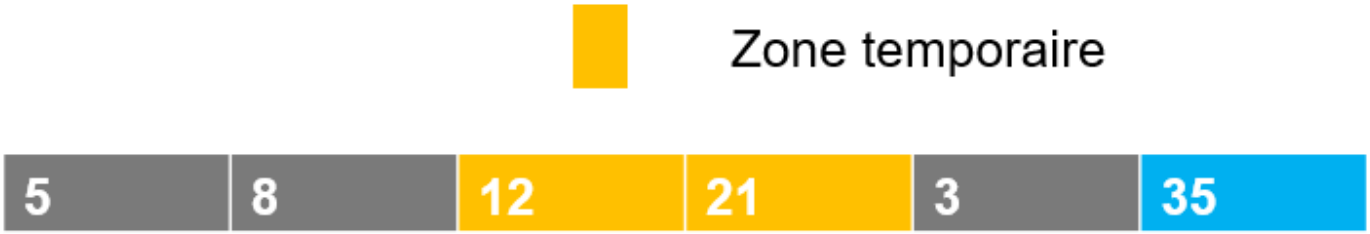
Simulation (6)



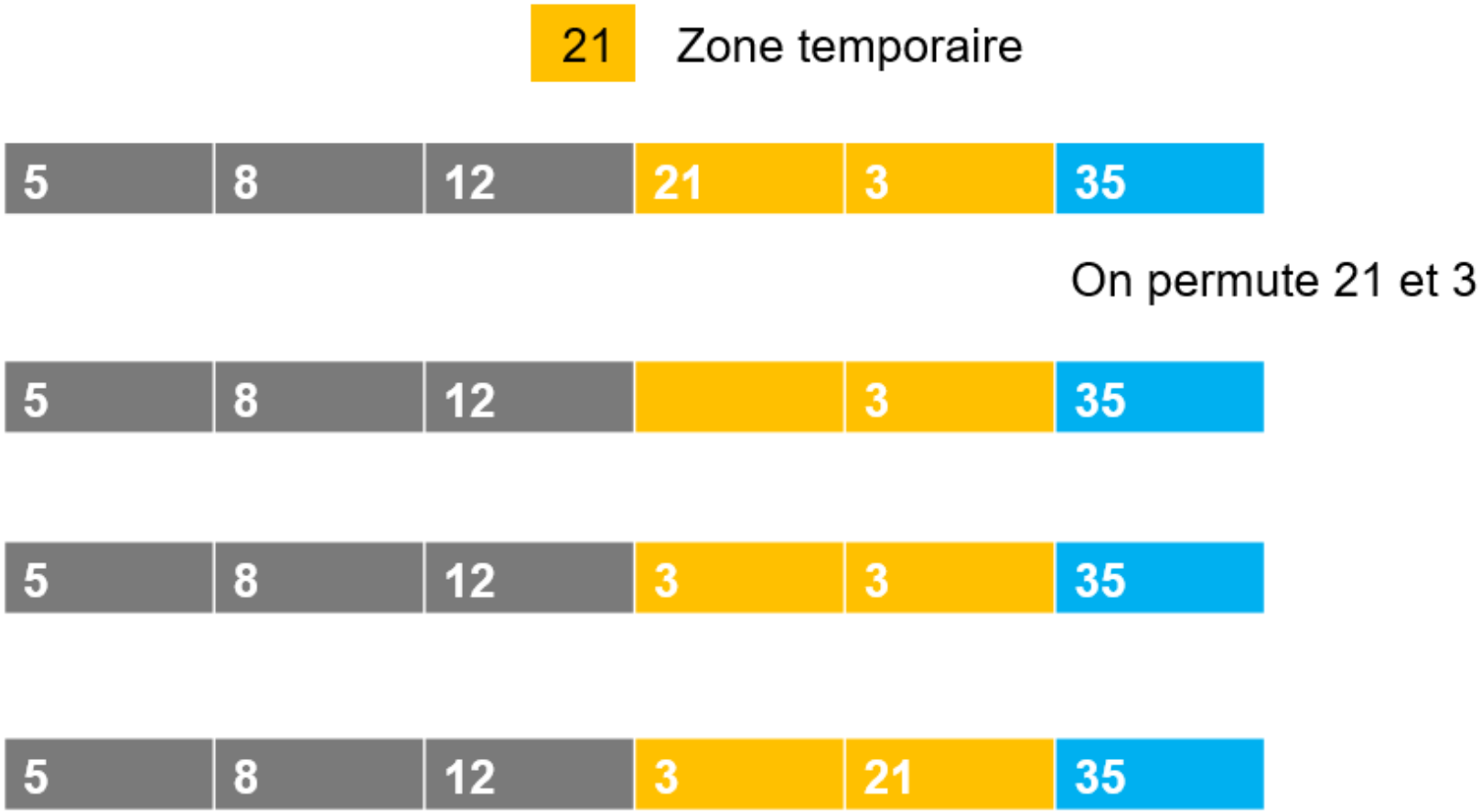
Simulation (7)



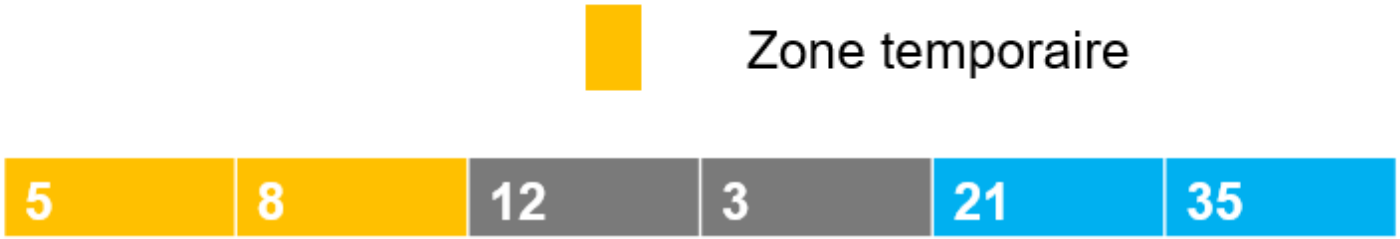
Simulation (8)



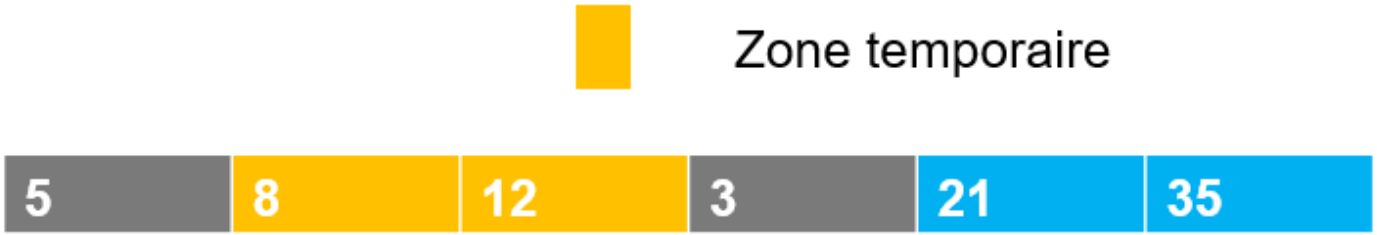
Simulation (9)



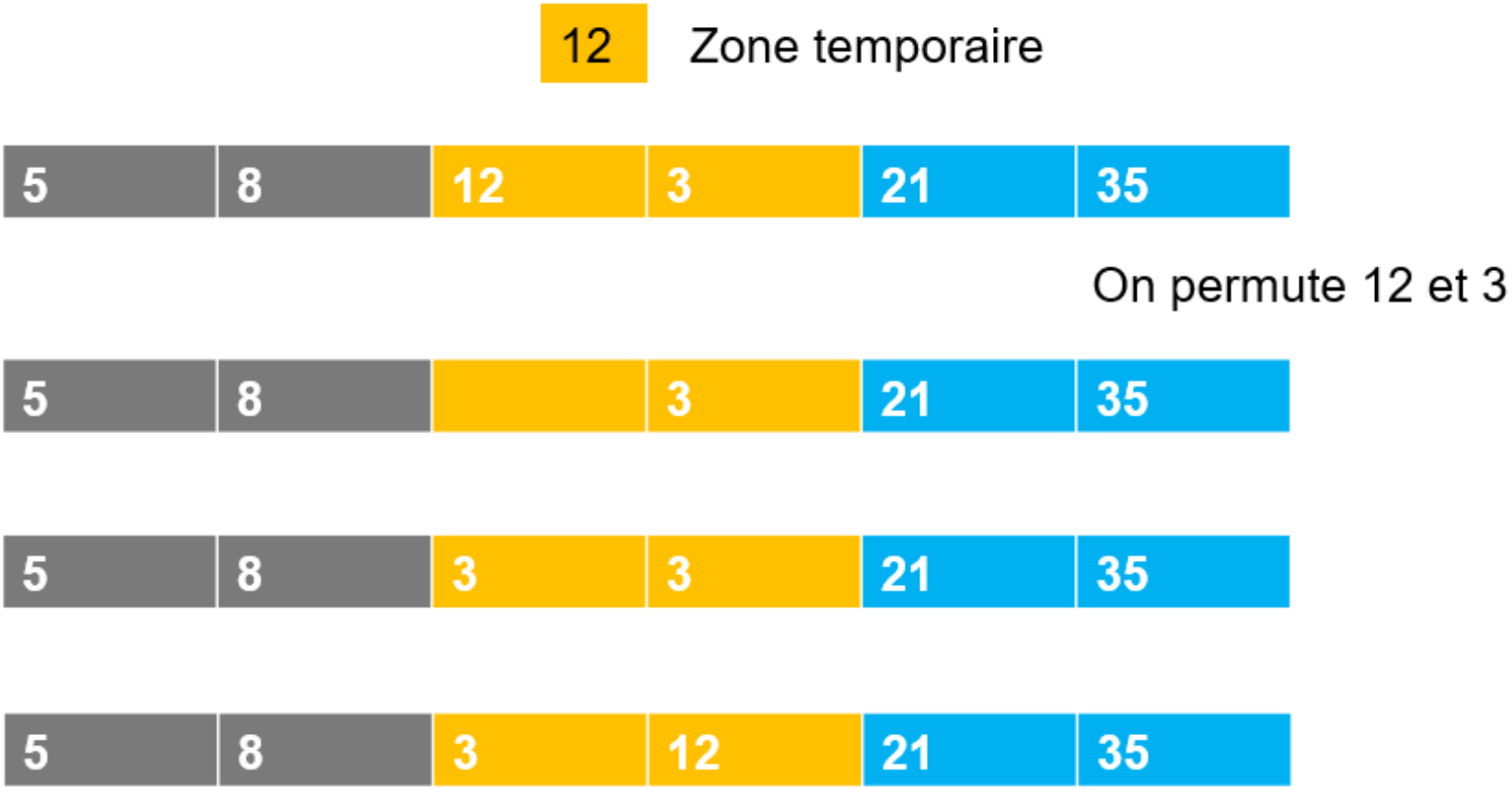
Simulation (10)



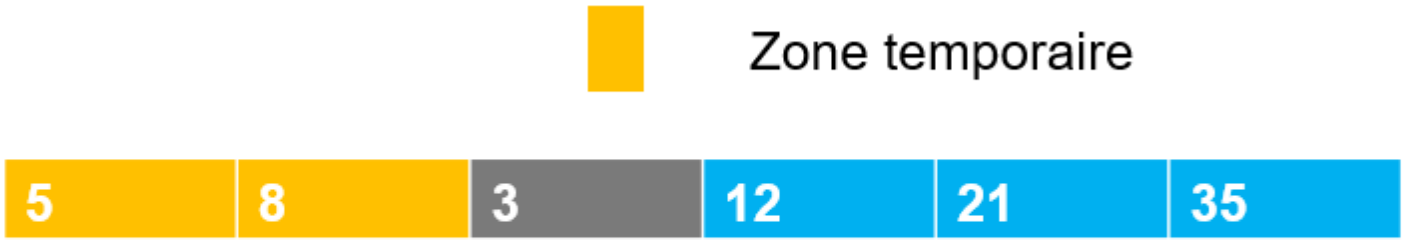
Simulation (11)



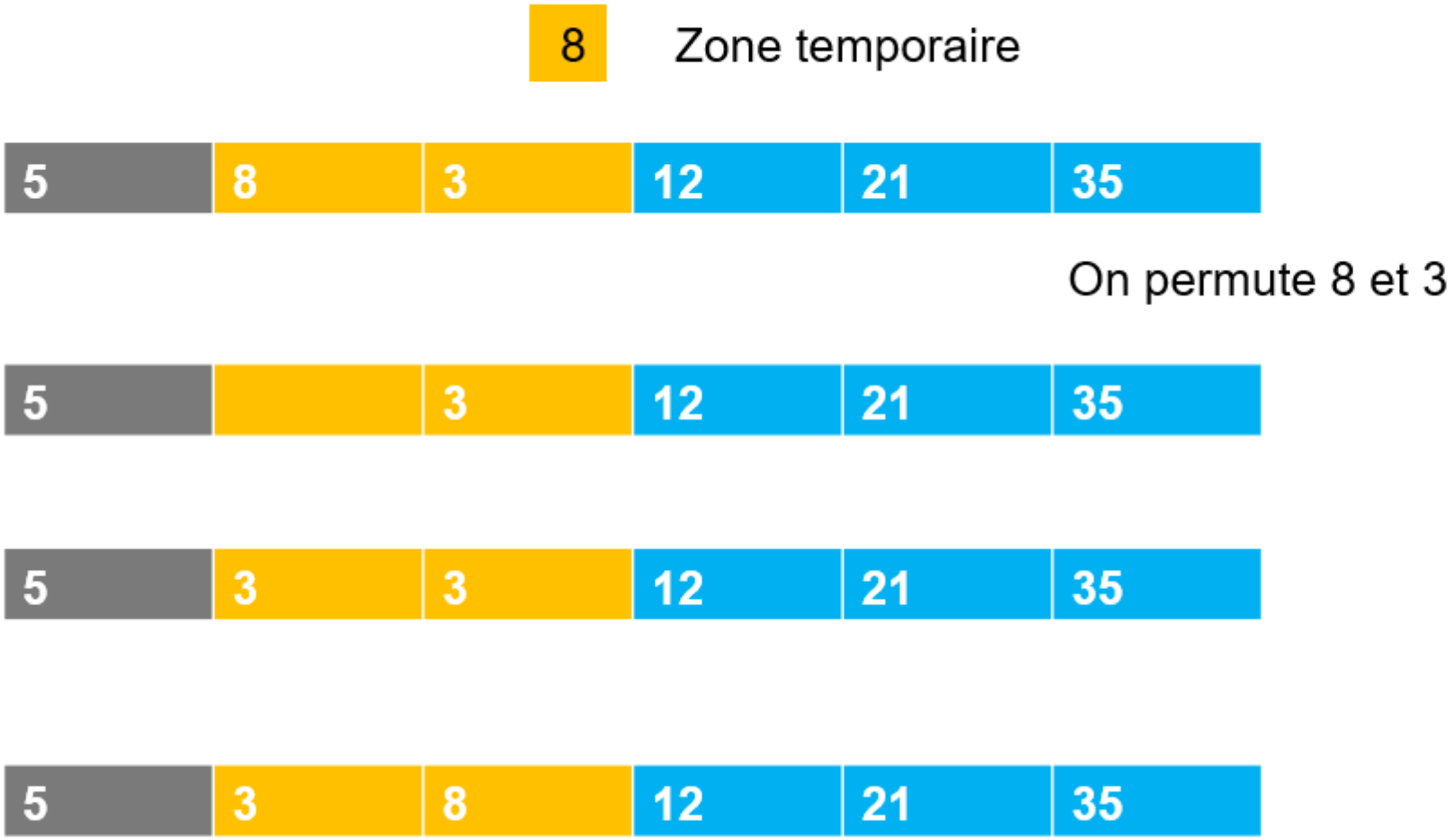
Simulation (12)



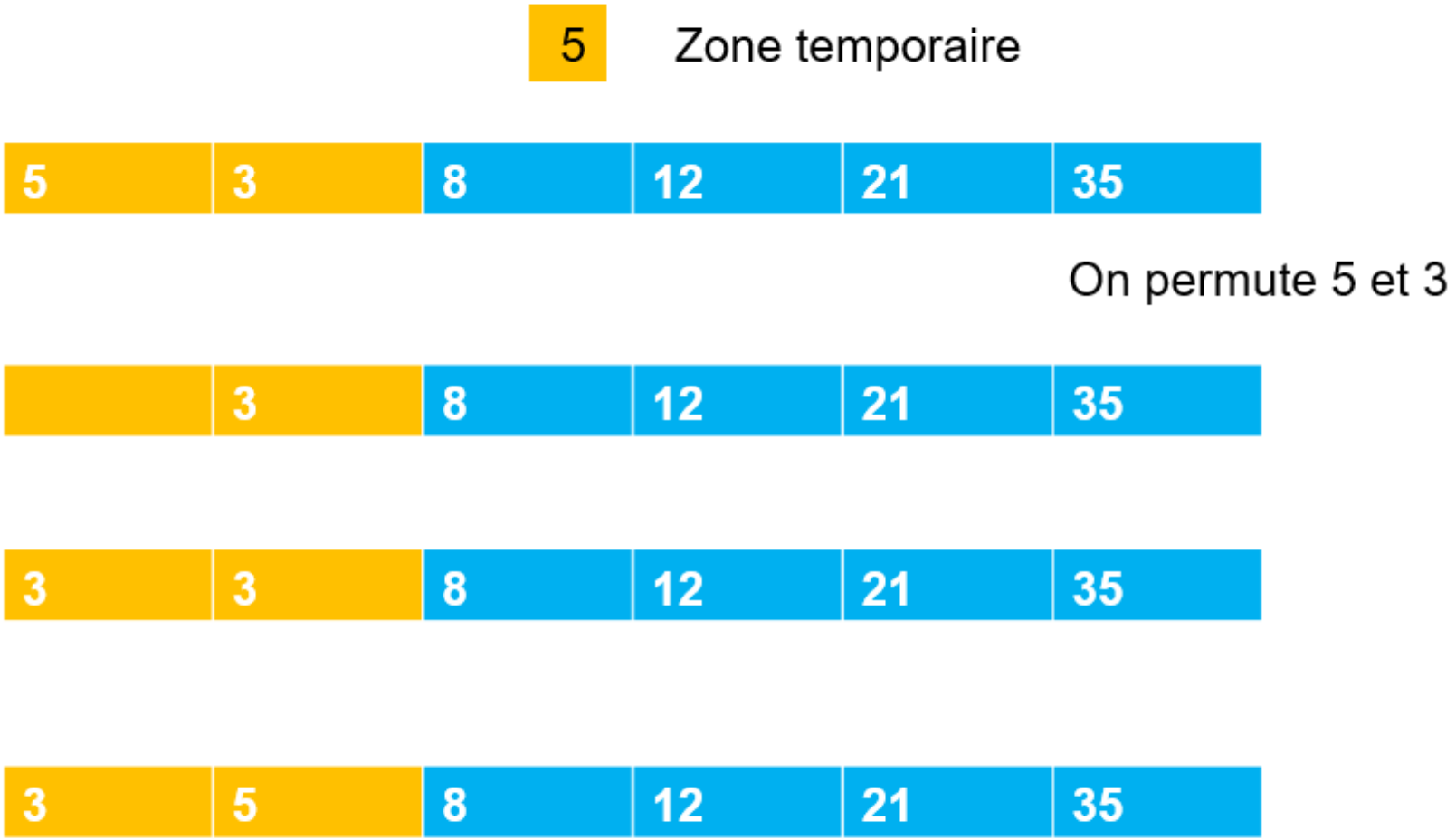
Simulation (13)



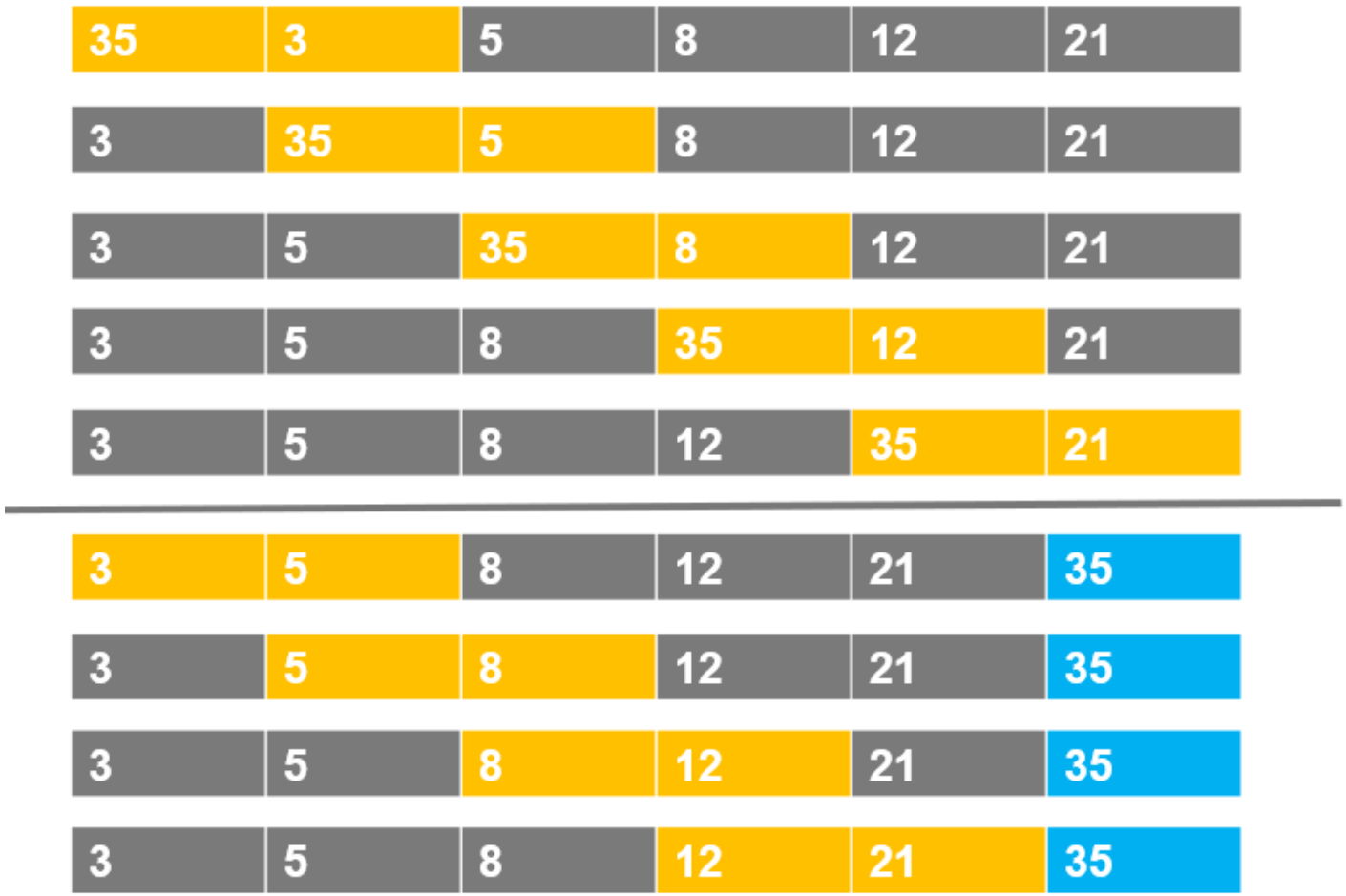
Simulation (14)



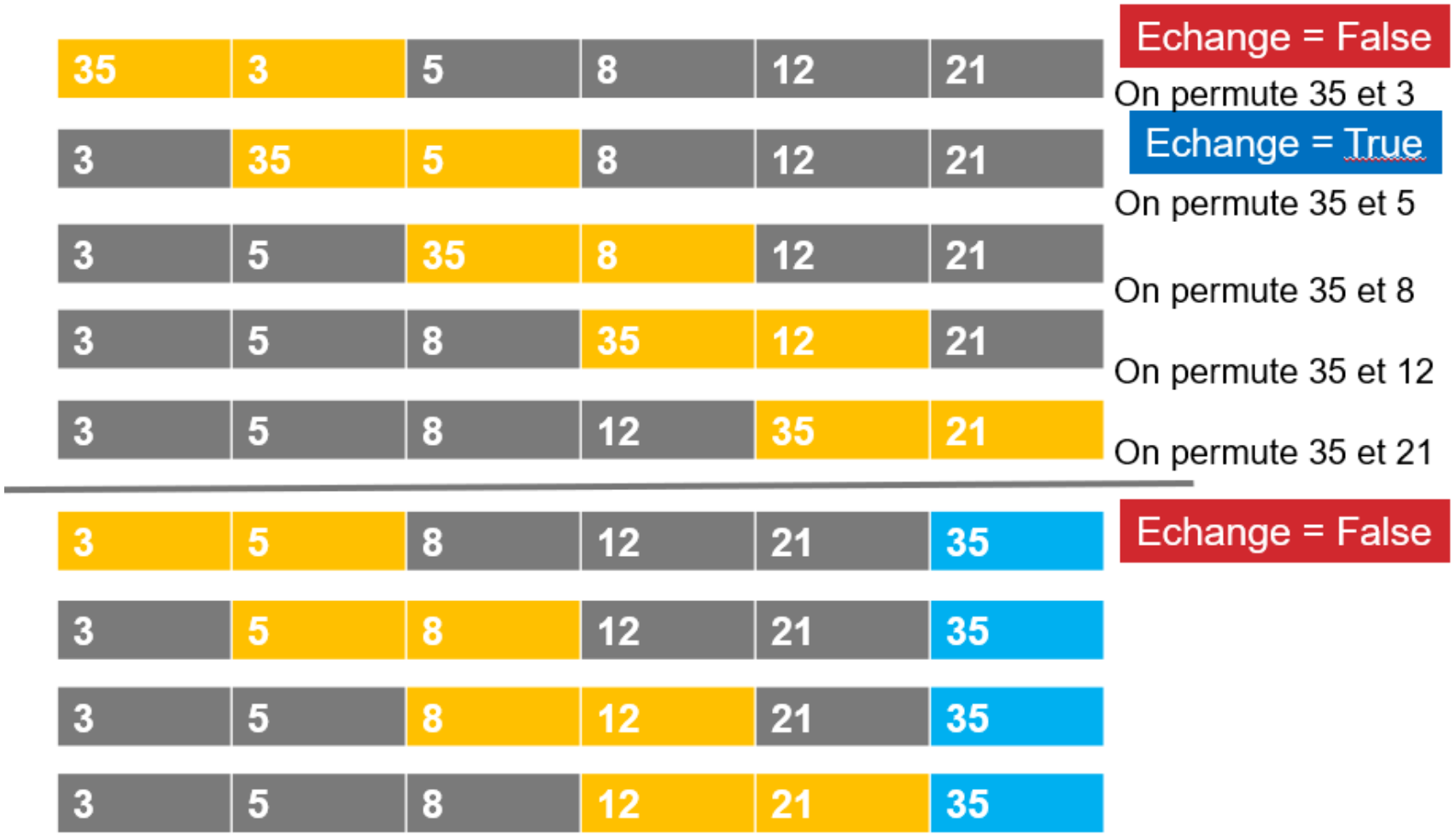
Simulation (15)



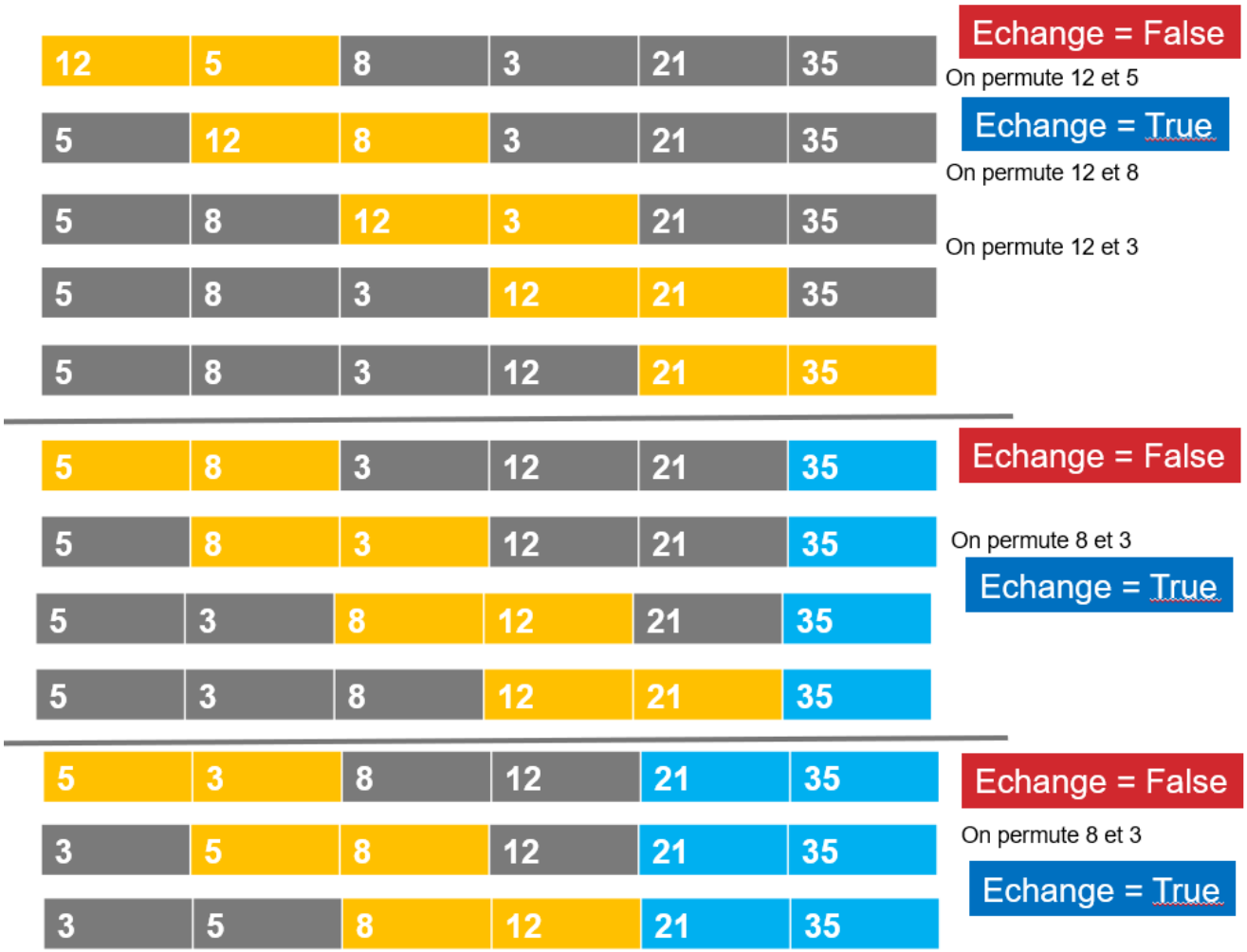
Simulation - Autre exemple (1)



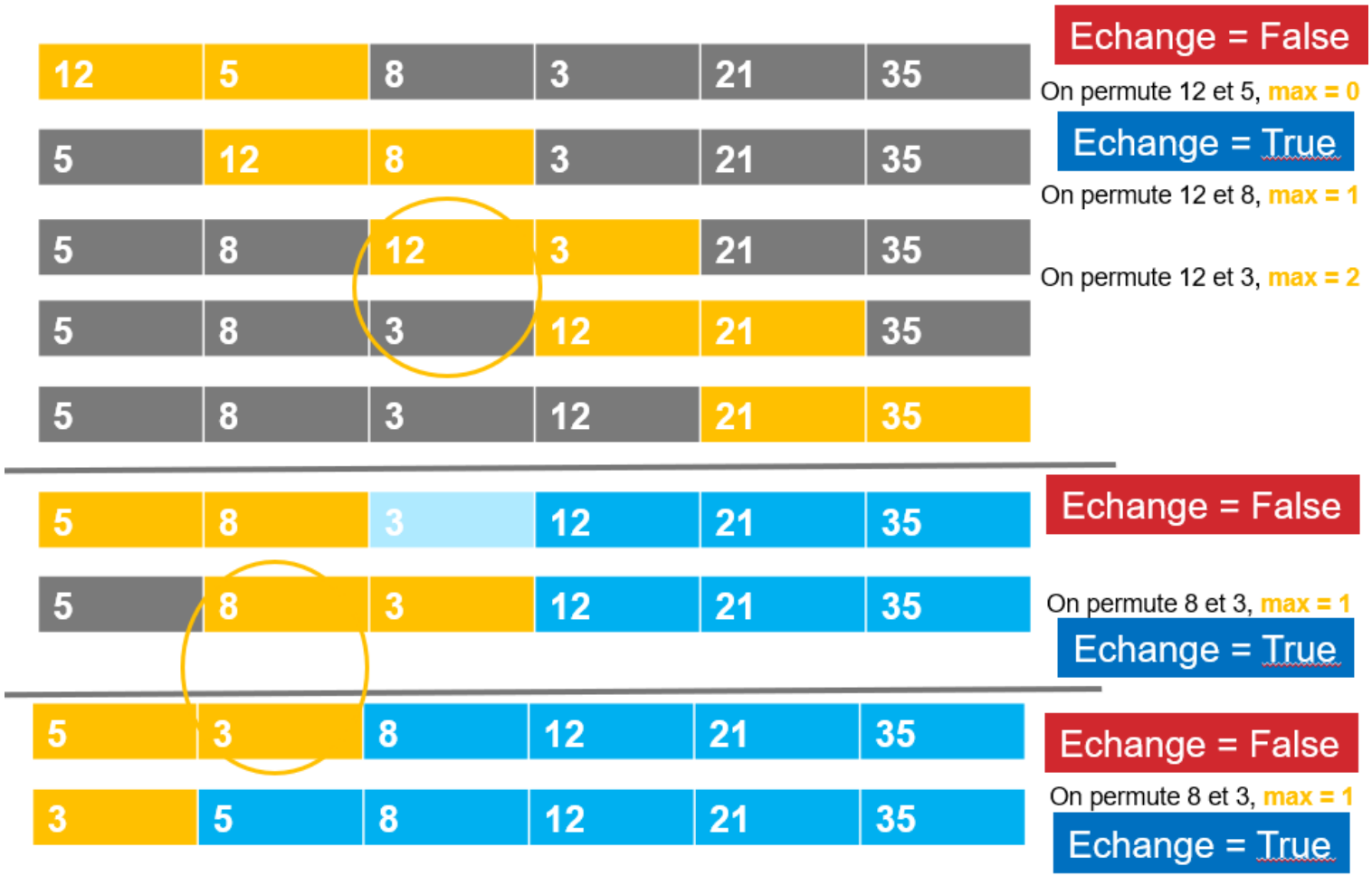
Simulation - Autre exemple (1)



Simulation - Autre exemple (2)



Simulation - Autre exemple (2)



Algorithme

Algorithm 6 Tri à bulles

```
nbelts ← len(t) − 1
echange ← True
tant que echange == True faire
    echange ← False
    max ← nbelts
    pour i allant de 0 à max faire
        si t[i + 1] < t[i] alors
            Permuter t[i] et t[i + 1]
            echange ← True
            nbelts = i
```

Complexité algorithme de tri à bulles

Le nombre d'échanges dépend plus fortement de l'ordre initial des éléments dans le tableau :

- Meilleur des cas : une comparaison à chaque insertion, donc $n-1$ comparaisons
- Moyenne : $n-1+\frac{n*(n-1)}{4}$
- Pire des cas : $\frac{n*(n-1)}{2}$

La complexité reste en $\mathcal{O}(n^2)$

Version Python

A faire en TD...

Principe

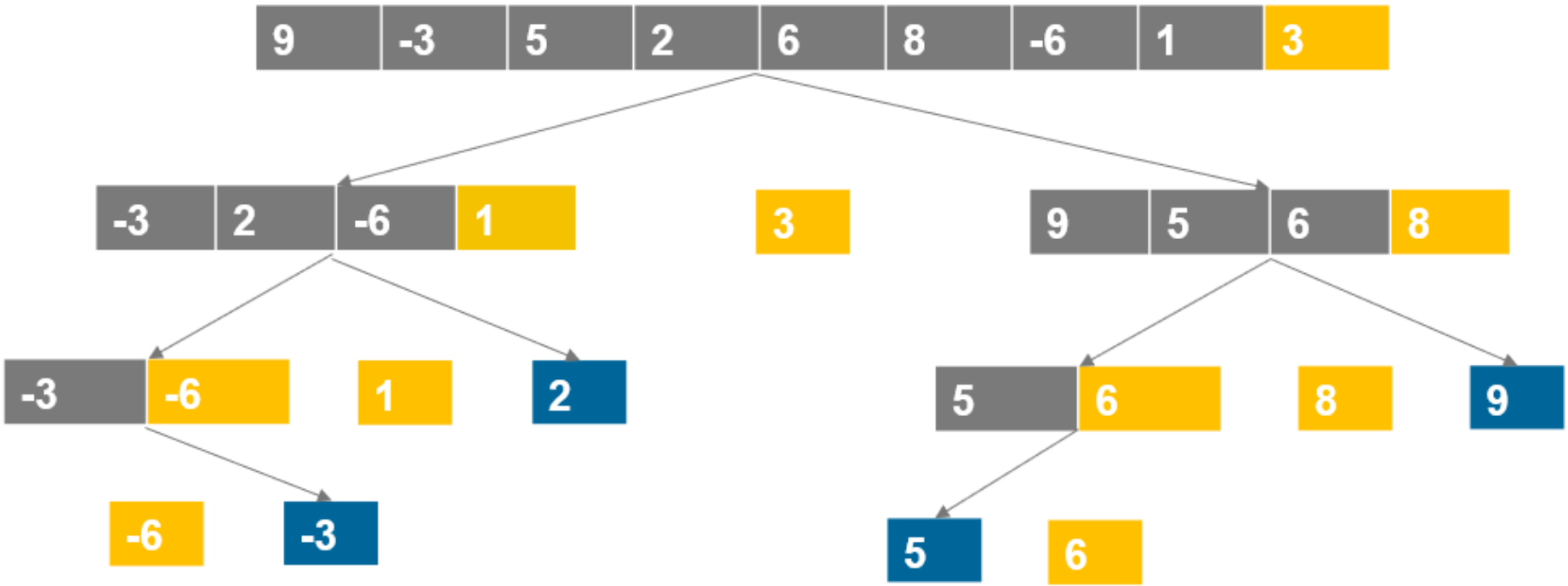
Tri récursif

- On choisit une valeur pivot
- On sépare dans le tableau les éléments inférieurs à cette valeur et les éléments supérieurs
- On trie ensuite chacune des deux parties du tableau en suivant le même principe

A voir : `https://interstices.info/les-algorithmes-de-tri/`

Ou : `https://marcarea.com/weblog/2019/02/06/quelques-algorithmes-de-tri-en-python`

Simulation



Version Python

```
def quickSort(t):
    'Tri rapide'
    if len(t) < 2 :
        return t
    else :
        pivot = t[-1]
        gauche = []
        droite = []
        for i in range(len(t)-1):
            if t[i] <= pivot :
                gauche.append(t[i])
            else :
                droite.append(t[i])
        return quickSort(gauche) + [pivot] + quickSort(droi
```

Problème

Complexité spatiale élevée :

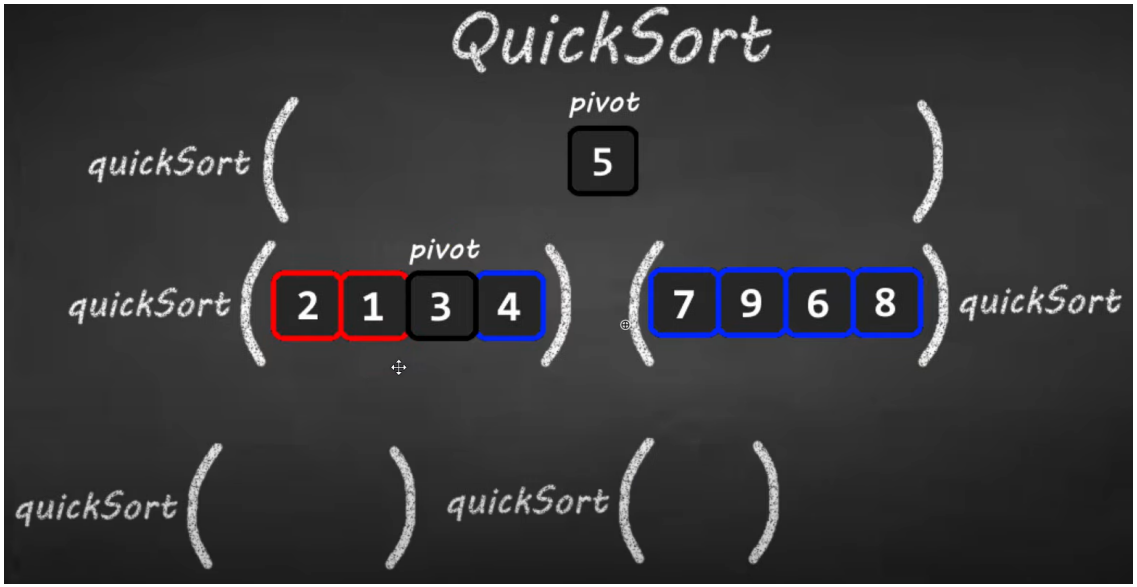
Ecrite ainsi, cette version n'est pas efficace

- À chaque passage dans la fonction, des nouvelles instances de tableaux sont créés au moment de la partition et stockées dans la pile d'exécution

Meilleure solution - Principe

Pour une meilleure complexité algorithmique :

- Méthodes de partition comme celle de Lomuto avec mutation du tableau en entrée



A voir : `https://youtu.be/Vtckgz38QHs`
`https://www.youtube.com/watch?v=MZaf_9IZCrc&t=155s`

Meilleure version - Python

```python
def triRapide(t, premier = 0, dernier = -1):
    'Tri rapide'
    if dernier == -1 :
        dernier = len(t)-1
    if premier < dernier :
        ipivot = partition(t, premier, dernier)
        triRapide(t, premier, ipivot - 1)
        triRapide(t, ipivot + 1, dernier)
    return t
```

```python
def partition (t, premier, dernier) :
    pivot = t[dernier]
    i = premier-1
    for j in range(premier, dernier) :
        if t[j] <= pivot :
            if i != j :
                i += 1
                permute(t, i, j)
    i += 1
    t[dernier] = t[i]
    t[i] = pivot
    return i
```

Complexité (1)

La partie du tri la plus sensible est le choix du pivot

- Le dernier
- Au milieu
- Aléatoirement :
  - Le choix peut être catastrophique : si le pivot est à chaque choix le plus petit élément du tableau, alors le tri rapide dégénère en tri par sélection

Complexité :

- Meilleur des cas : $\mathcal{O}(n \log n)$
- Moyenne : $\mathcal{O}(n \log n)$
- Pire des cas : $\mathcal{O}(n^2)$

Astuces

Il existe des astuces pour rendre le cas dégénéré le plus improbable possible, ce qui rend **cette méthode la plus rapide en moyenne parmi toutes celles utilisées**

Source : `https://interstices.info/les-algorithmes-de-tri/`

Complexité (2)

	N log N	N^2
10^3	~ 10^4	10^6
10^6	~ 20 10^6	10^{12}

Fonctions de tri en Python (1)

Fonction `sorted()`

- Retourne un tableau trié sans modifier le tableau initial
- Exemple :

```
>>> t = [9, -3, 5, 2, 6, 8, -6, 1, 3]
>>> print(sorted(t))
[-6, -3, 1, 2, 3, 5, 6, 8, 9]
>>> t
[9, -3, 5, 2, 6, 8, -6, 1, 3]
```

Fonctions de tri en Python (2)

Méthode `t.sort()`

- Modifie le tableau
- Exemple :

```
>>> t.sort()
>>> t
[-6, -3, 1, 2, 3, 5, 6, 8, 9]
```

Fonctions de tri en Python (3)

`sorted()` et `t.sort()`

- L'algorithme de tri utilisé en Python est appelé timsort, inventé par Tim Peters en 2002.
- C'est un algorithme dérivé du tri fusion et du tri par insertion
- L'idée principale : dans un ensemble de données, une partie est déjà triée et il faut en tirer un avantage.
- **Complexité :**
  - Meilleur des cas : $\mathcal{O}(n)$
  - Moyenne : $\mathcal{O}(n \log n)$
  - Pire des cas : $\mathcal{O}(n \log n)$

Fonctions de tri en Python (3)

Tri de dictionnaires par clé

```
>>> peintres = [{"nom" : "Da Vinci", "naissance":1452, "mort":1519},
...             {"nom" : "Michelangelo", "naissance":1475, "mort":1564},
...             {"nom" : "Raphaël", "naissance":1483, "mort":1520},
...             {"nom" : "Bellini", "naissance":1430, "mort":1516}]
...
>>> def nom(x) :
...     return x["nom"]
...
>>> def naissance(x) :
...     return x["naissance"]
...
>>> def mort(x) :
...     return x["mort"]
...
>>> peintres.sort(key=nom)
>>> peintres
[{'nom': 'Bellini', 'naissance': 1430, 'mort': 1516}, {'nom': 'Da Vinci', 'naissance': 1452, 'mort': 1519}, {'nom': 'Michelangelo', 'naissance': 1475,
  'mort': 1564}, {'nom': 'Raphaël', 'naissance': 1483, 'mort': 1520}]
>>> peintres.sort(key=naissance)
>>> peintres
[{'nom': 'Bellini', 'naissance': 1430, 'mort': 1516}, {'nom': 'Da Vinci', 'naissance': 1452, 'mort': 1519}, {'nom': 'Michelangelo', 'naissance': 1475,
  'mort': 1564}, {'nom': 'Raphaël', 'naissance': 1483, 'mort': 1520}]
```

Questions...