HESTIM
ENGINEERING & BUSINESS SCHOOL

Algorithmes de Tri et de Recherche

Filière : Ingénierie Informatique & Intelligence Artificielle

Par : Youssef Ouassit

Plan

- Introduction
- Tri par fusion (Merge Sort)
  - Pseudo-code et illustration de l'algorithme
  - Avantages et inconvénients
- Tri rapide (Quick Sort)
  - Pseudo-code et illustration de l'algorithme
  - Avantages et inconvénients
- Autres algorithmes : Count Sort, Radix Sort, Tim, Shell...

Introduction

Définition :

Un algorithme de tri est en Informatique ou en mathématique, un algorithme qui permet de trier une collection d'objets selon un ordre bien déterminée. Les objets à trier font donc parti d'un ensemble muni d'une relation d'ordre.

Introduction

La vitesse de tri est un facteur critique dans de nombreuses applications :

- **Bases de Données:** Dans les systèmes de gestion de bases de données, le tri est essentiel pour exécuter des requêtes efficacement.
- **Traitement d'Images:** Les applications de traitement d'images, telles que la recherche d'images similaires ou la segmentation d'images, peuvent tirer parti de l'utilisation d'algorithmes de tri avancés
- **Calcul Scientifique:** Dans le calcul scientifique et l'analyse de données, le tri est souvent utilisé pour organiser des ensembles de données avant l'application d'autres algorithmes.
- **Algorithmes de Compression:** Dans les algorithmes de compression de données, tels que ceux utilisés dans les formats de fichiers audio et vidéo, un tri efficace peut réduire la taille des données en organisant les informations de manière optimale.
- **Algorithmes de Machine Learning:** Lors de la préparation de données pour des modèles d'apprentissage automatique, le tri peut être utilisé pour organiser les caractéristiques ou les échantillons.

Introduction

Algorithmes de tri quadratiques:

Tri à bulles:

1. Parcourez la liste d'éléments à trier, en comparant chaque élément avec l'élément adjacent à sa droite.

2. Si l'élément actuel est plus grand (dans le cas d'un tri croissant) que l'élément adjacent, échangez-les.

3. Continuez à parcourir la liste en répétant les comparaisons et les échanges jusqu'à ce qu'il n'y ait plus d'échanges nécessaires lors d'un passage complet à travers la liste.

4. Répétez cette opération jusqu'à ce que la liste soit entièrement triée.

Introduction

Algorithmes de tri quadratiques :

Tri à bulles:

```
def tri_bulle(L) :
    for i in range(len(L)-1, 0, -1 ) :
        for j in range(0, i):
            if L[j]>L[j+1] :
                L[j],L[j+1] = L[j+1], L[j]
    return L
```

Paramètre de complexité : n=len(L)

$$T(n) = \sum_{i=1}^{n-1} 6i + 1$$
$$= 6\sum_{i=1}^{n-1} i + 1$$
$$= 6(n-1)(n)/2 + 1$$
$$= 3n^2 - 3n + 1$$

Complexité quadratique: $O(n^2)$

Algorithmes de tri quadratiques :

Tri par sélection:

1. Initialement, la liste est divisée en deux parties : la partie triée (initialement vide) et la partie non triée (contenant tous les éléments non triés).

2. À chaque étape, l'algorithme parcourt la partie non triée pour trouver l'élément le plus petit (ou le plus grand, selon l'ordre de tri souhaité).

3. Une fois que l'élément le plus petit a été trouvé, il est échangé avec l'élément situé à la fin de la partie triée, de sorte que cet élément devienne le dernier élément de la partie triée.

4. Répétez ces étapes jusqu'à ce que la liste entière soit triée.

Algorithmes de tri quadratiques :

Tri par sélection:

```
def tri_selection(T) :
    for i in range(len(T)-1) :
        posmin=i
        for j in range(i+1,len(T)) :
            if T[j]<T[posmin] :
                posmin=j
        T[i],T[posmin]=T[posmin],T[i]
    return T
```

Paramètre de complexité : n=len(T)

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{n-2}\left(3 + \left(\sum_{j=i+1}^{n-1} 2\right)\right) \\
&= \sum_{i=0}^{n-2} 3 + \sum_{i=0}^{n-2}\left(\sum_{j=i+1}^{n-1} 2\right) \\
&= 3(n-1) + \sum_{i=0}^{n-2}\left(\sum_{j=1}^{n-i-1} 2\right) \\
&= 3(n-1) + 2\sum_{i=0}^{n-2}(n-i-1) \\
&= 3(n-1) + 2\left(\sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1\right) \\
&= 3(n-1) + 2(n(n-1) - (n-1)(n)/2 - (n-1)) = n^2 - 5n - 2
\end{aligned}
$$

Complexité quadratique: $O(n^2)$

Algorithmes de tri quadratiques :

Tri par insertion:

1. Initialement, la première élément de la liste est considéré comme la partie triée (car un élément unique est toujours trié).

2. Ensuite, l'algorithme passe en revue la partie non triée de la liste, un élément à la fois.

3. À chaque étape, l'algorithme prend l'élément actuel de la partie non triée et le compare à chaque élément de la partie triée, de droite à gauche, pour trouver la position appropriée où l'élément actuel devrait être inséré.

4. Lorsque la position appropriée est trouvée, l'élément actuel est inséré à cet endroit, et les éléments suivants dans la partie triée sont décalés vers la droite pour faire de la place pour l'élément inséré.

5. Répétez ces étapes jusqu'à ce que tous les éléments de la partie non triée aient été insérés dans la partie triée.

Introduction

Algorithmes de tri quadratiques :

Tri par insertion:

```
def tri_insertion(T) :
    for i in range(1,len(T)) :
        k=i
        while k>0 and T[k]<T[k-1] :
            T[k], T[k-1] = T[k-1], T[k]
            k = k – 1
    return T
```

Paramètre de complexité : n=len(T)

$T(n) = \sum_{i=1}^{n-1}(1 + 10i)$

$T(n) = (n-1) + 10((n-1)(n)/2)$

$T(n) = 5n^2 - 4n - 6$

Complexité quadratique: $O(n^2)$

Introduction

Paradigme diviser pour régner :

Diviser: Si la taille des données est trop grande pour les traiter d'une façon directe alors les diviser en deux ou plusieurs sous-ensembles disjoints.

Appliquer récursivement: Utiliser l'approche diviser pour régner afin de résoudre les sous-problèmes associés aux sous-ensembles de données.

Régner: Prendre les solutions aux sous-problèmes et "fusionner" ces solutions afin d'obtenir la solution au problème initial.

Le cas de base pour la récursivité est le sous-problème trivial, souvent de taille 0 ou 1.

Tri par fusion

Principe :

Le principe de cet algorithme tend à adopter une formulation récursive :

- ✓On découpe les données à trier en deux parties plus ou moins égales
- ✓On trie les 2 sous-parties ainsi déterminées
- ✓On fusionne les deux sous-parties pour retrouver les données de départ

Tri par fusion

Exemple :

98	23	45	14	6	67	33	42

Tri par fusion

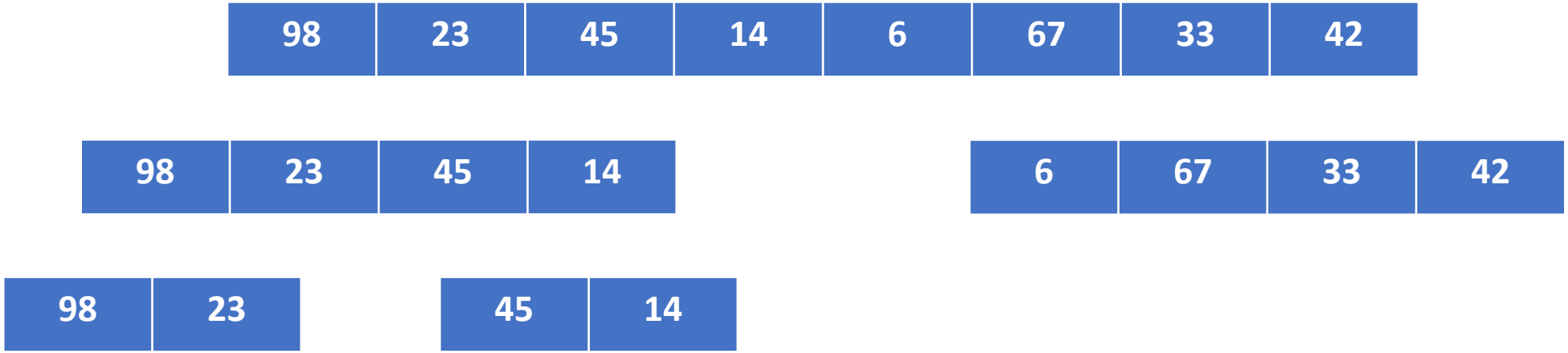
Exemple :

98	23	45	14	6	67	33	42

98	23	45	14		6	67	33	42

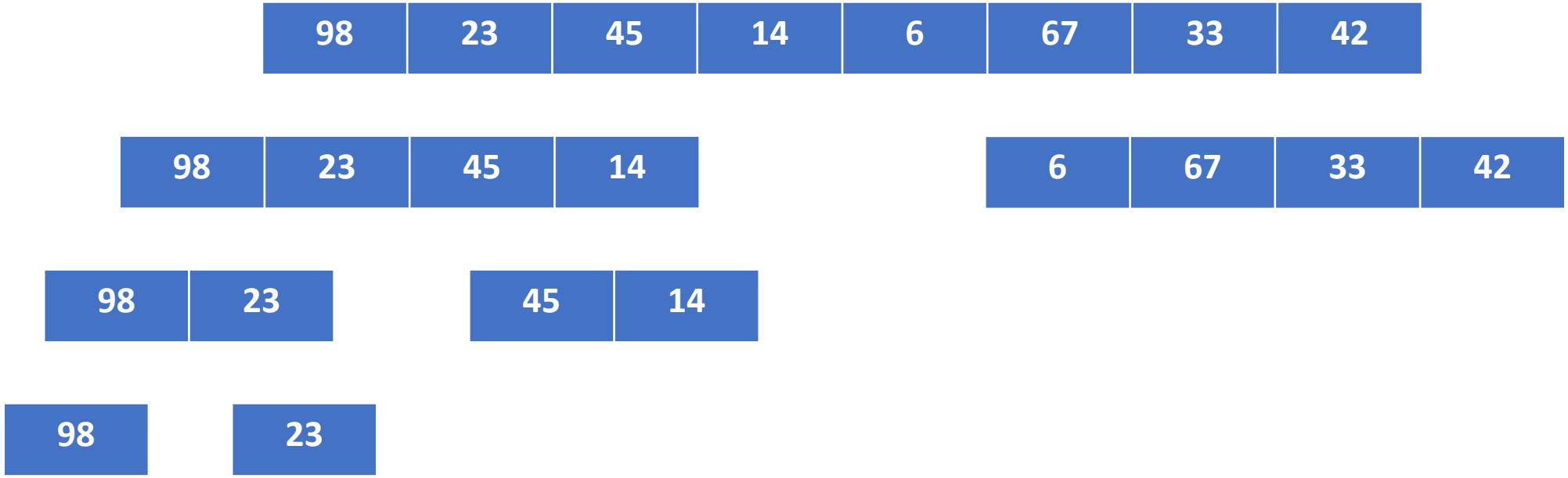
Tri par fusion

Exemple :



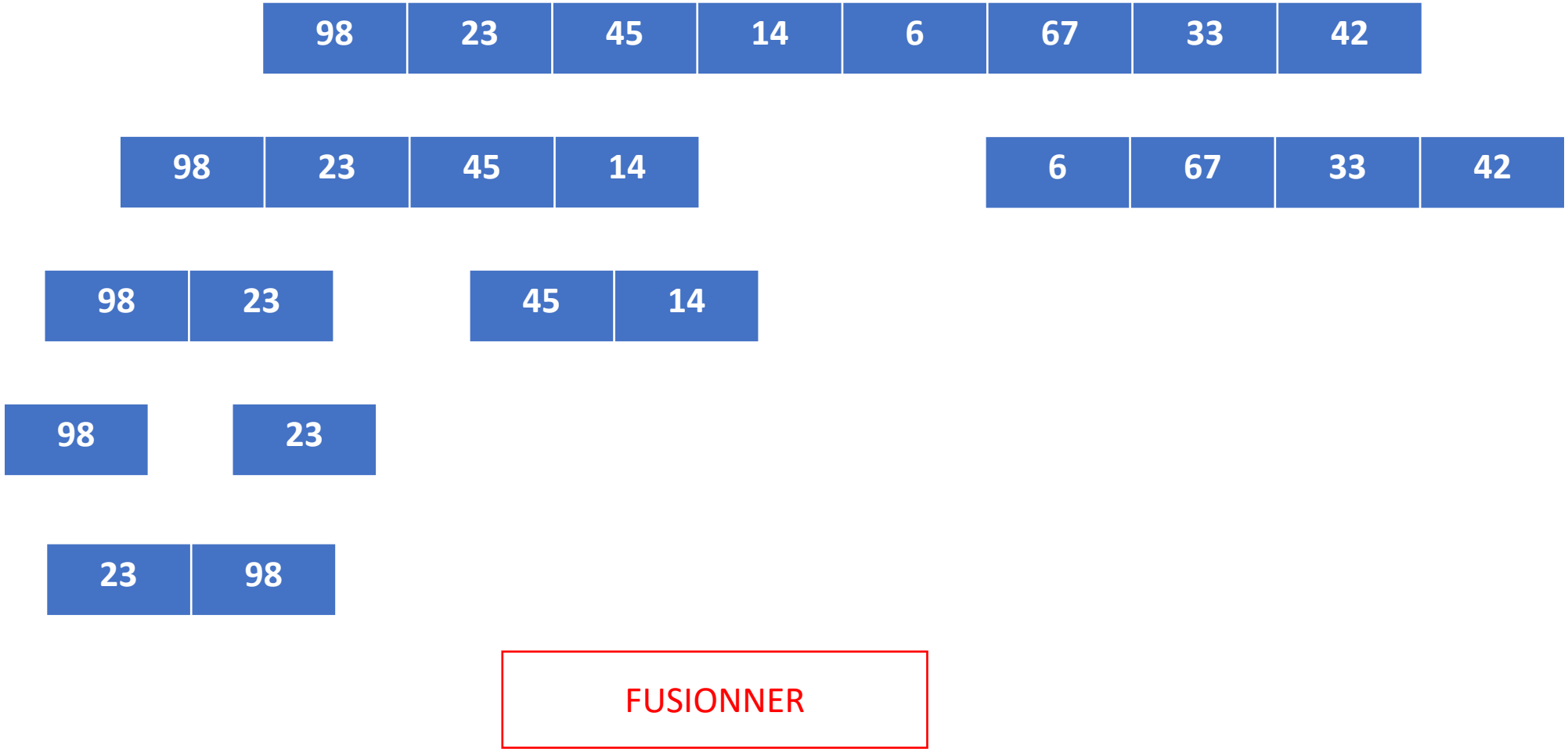
Tri par fusion

Exemple :



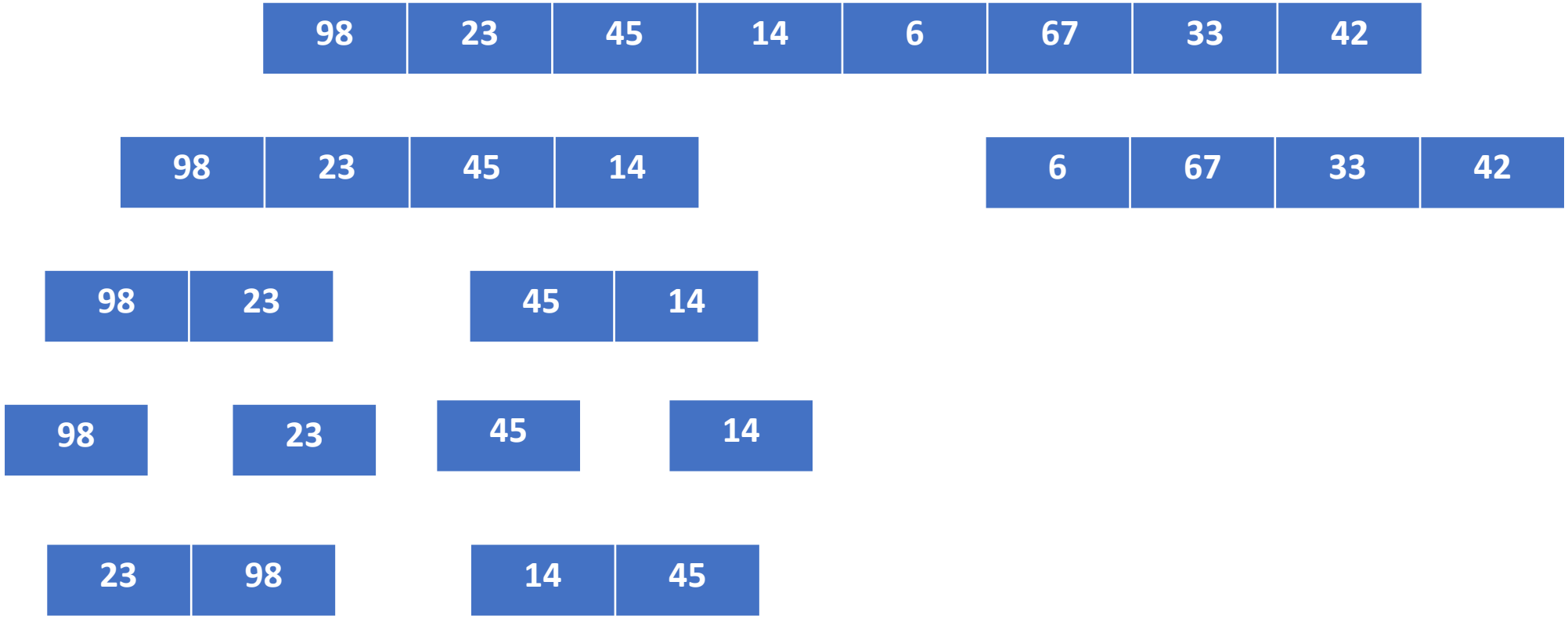
Tri par fusion

Exemple :



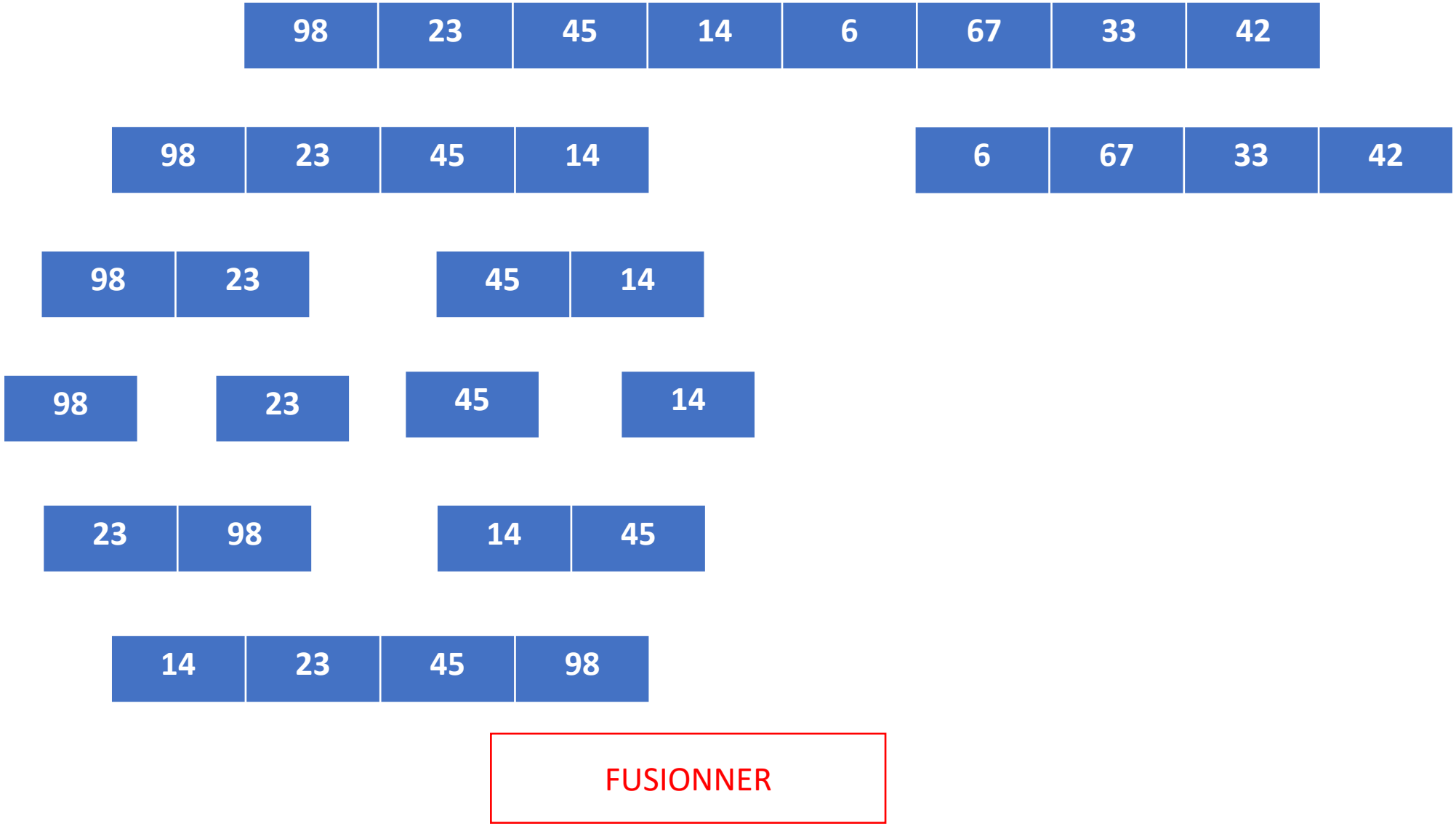
Tri par fusion

Exemple :



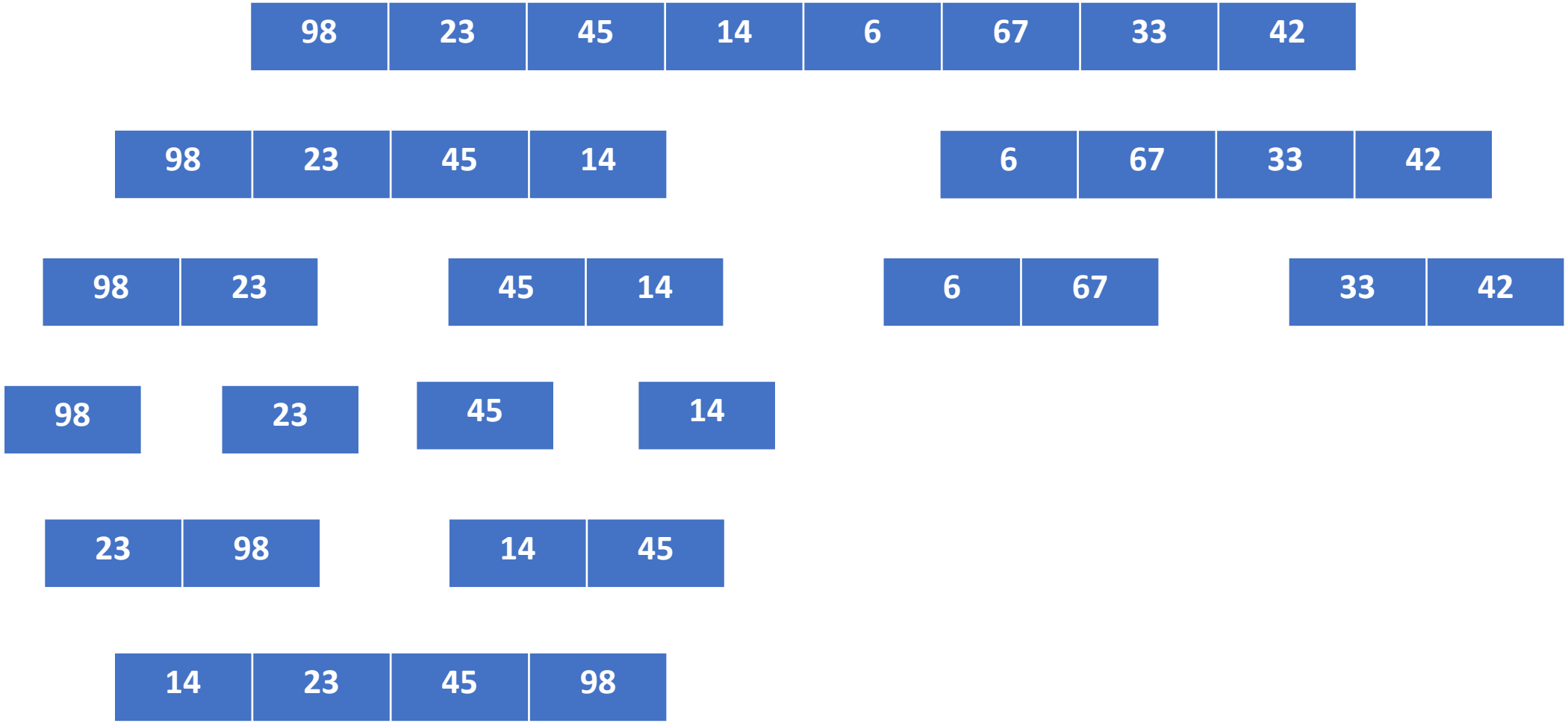
Tri par fusion

Exemple :



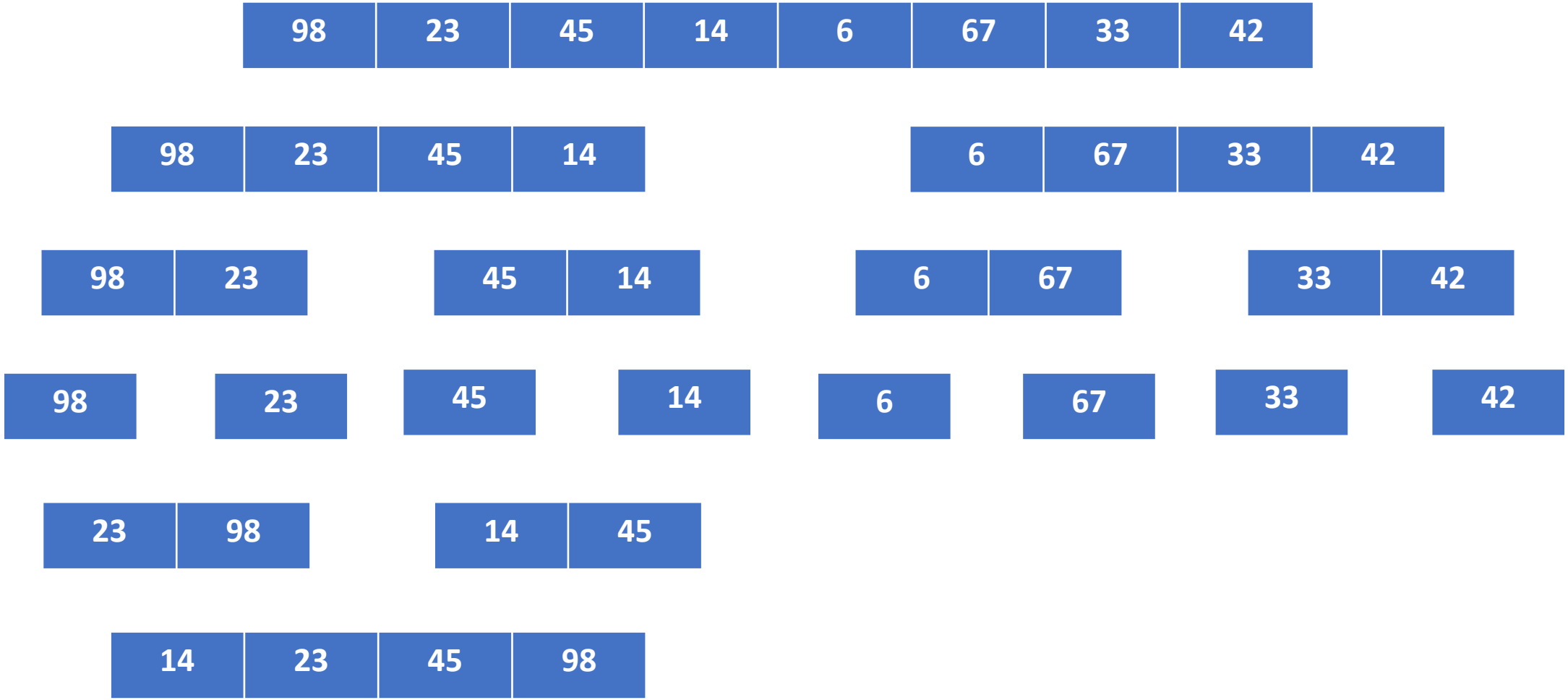
Tri par fusion

Exemple :



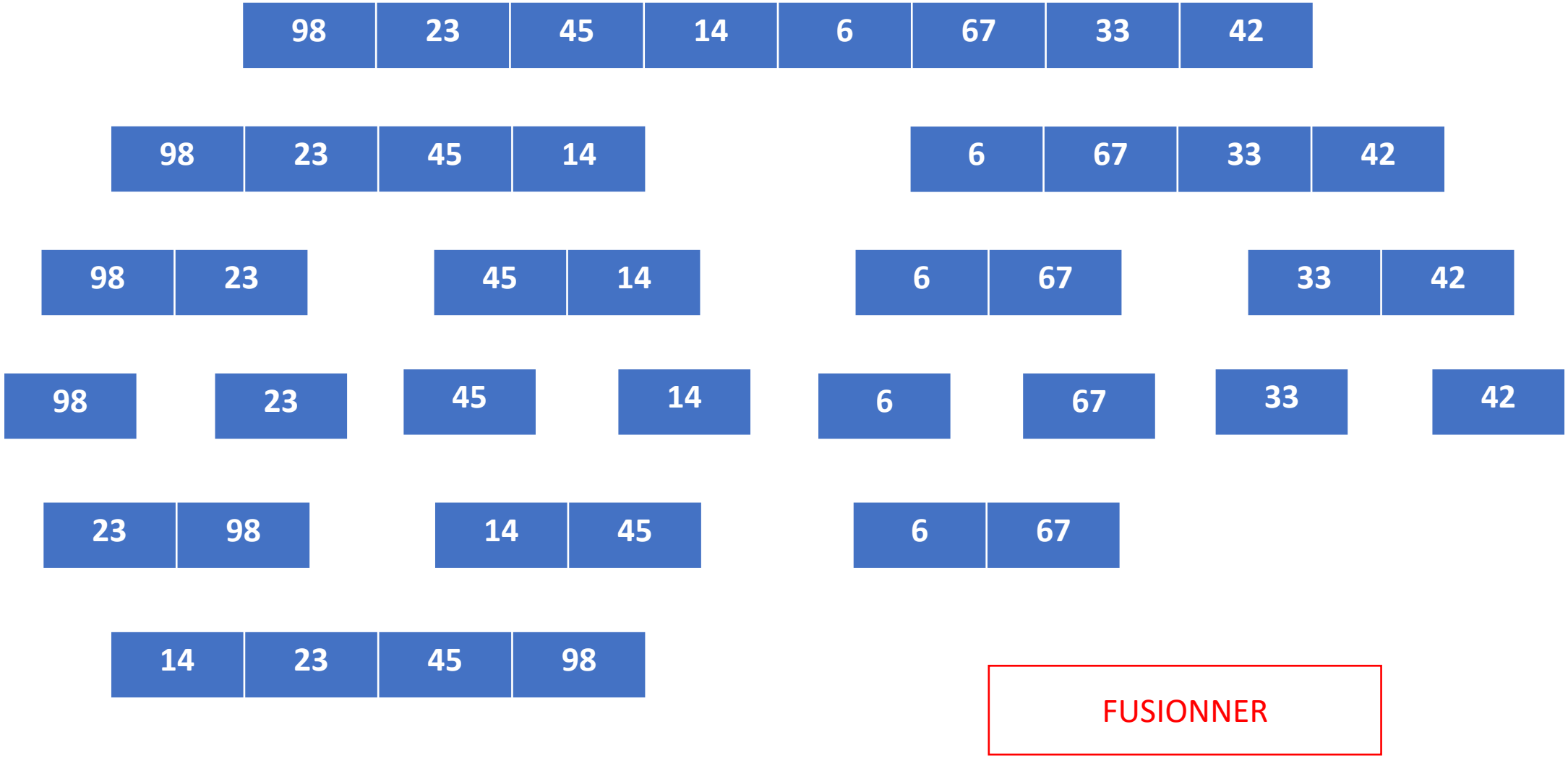
Tri par fusion

Exemple :



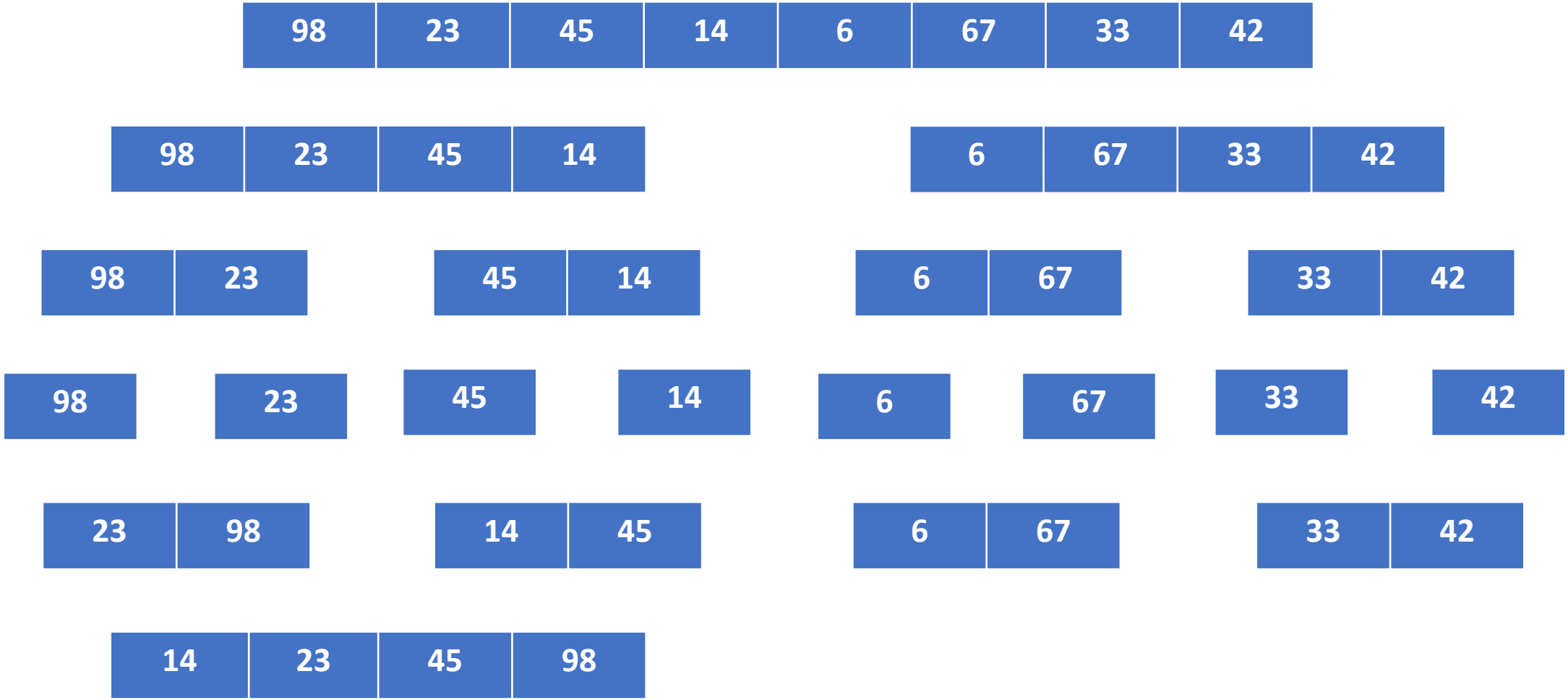
Tri par fusion

Exemple :



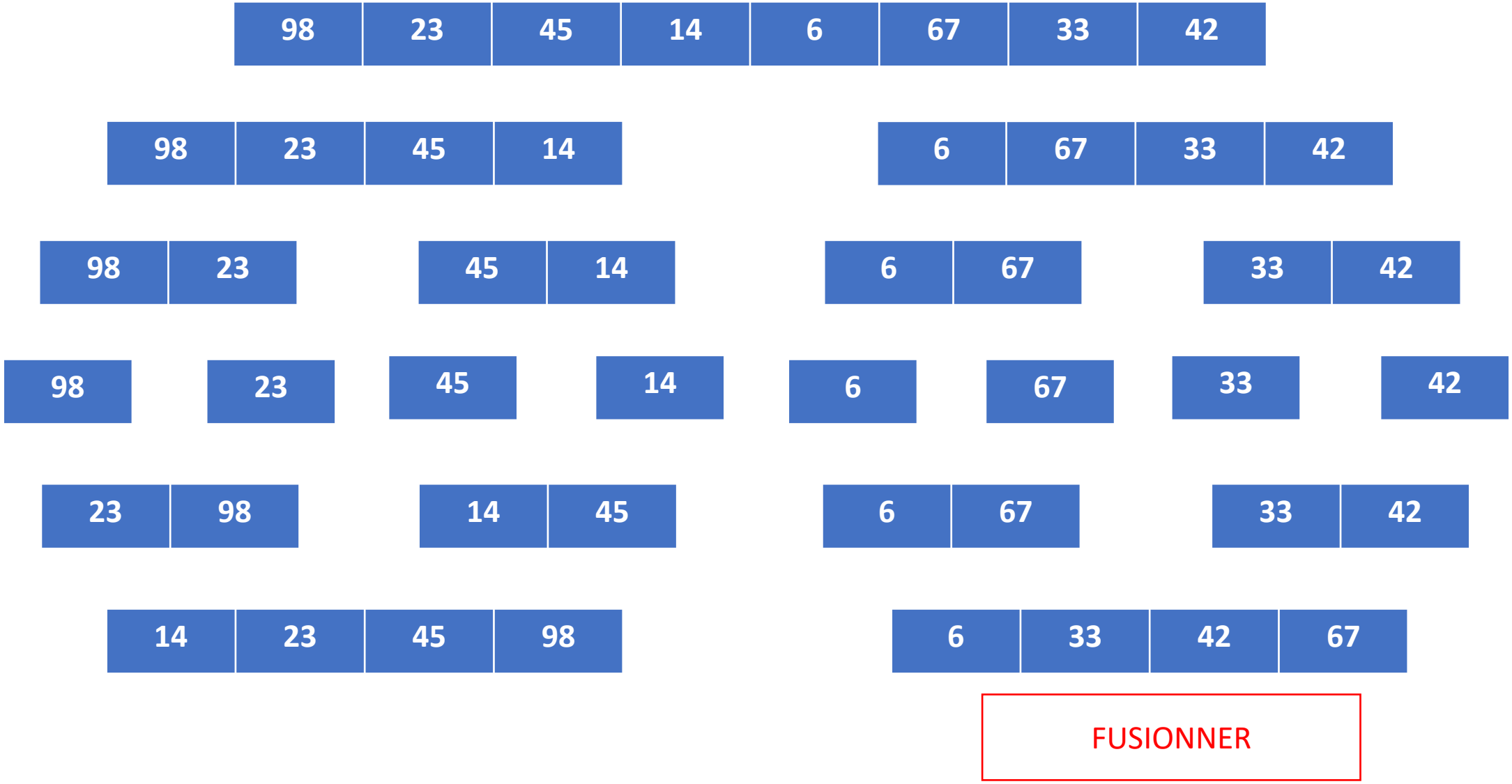
Tri par fusion

Exemple :



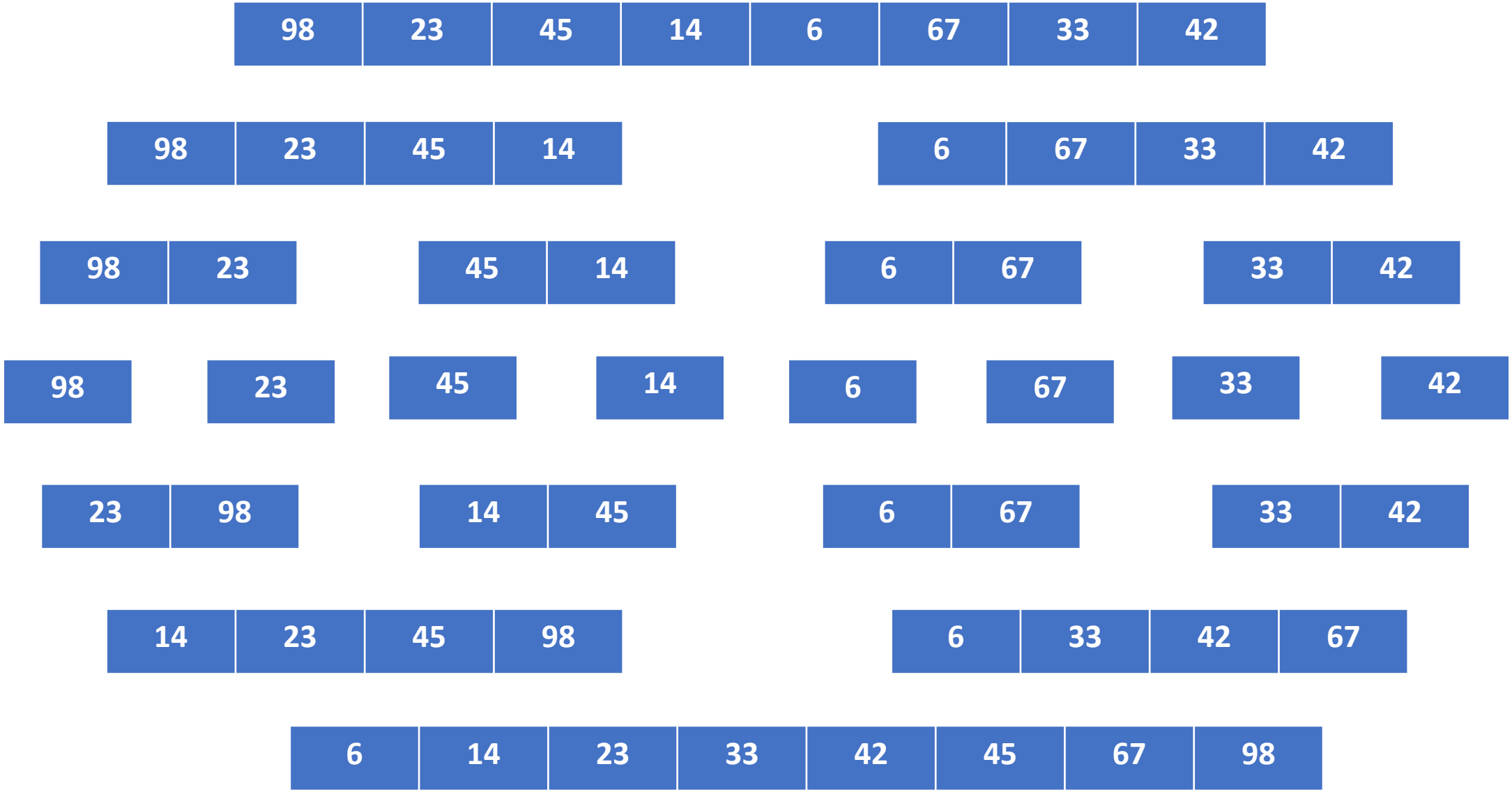
Tri par fusion

Exemple :



Tri par fusion

Exemple :



Tri par fusion

Algorithme :

```
TriFusion(liste)
    Si la taille de la liste est <= 1
        Retourner la liste non modifiée
    Diviser la liste en deux moitiés, gauche et droite
    gauche = TriFusion(gauche) // Tri récursif de la moitié gauche
    droite = TriFusion(droite) // Tri récursif de la moitié droite
    Fusionner les moitiés triées gauche et droite en une seule liste triée
    Retourner la liste fusionnée
```

Implémentation :

L'implémentation de cet algorithme repose essentiellement sur 2 fonctions :

fusionner(A, B) : Cette fonction prend en paramètre deux listes triées dans un ordre croissant et retourne une nouvelle liste triée qui contient les éléments de A et B fusionnés.

tri_fusion(L) : On vérifier si la liste L fournie en entrée n'est pas vide ou ne se résume pas à un seul élément, auquel cas, il n'y a rien à faire !
Si les tests sont concluants, on lancera la fonction de fusion sur 2 sous séquences récursives du tableau.

Complexité des fonctions récursives

```
def fusionner(A, B):
    R = []
    g, d = 0, 0
    while g<len(A) and d <len(B):
        if A[g] < B[d] :
            R.append(A[g])
            g += 1
        else :
            R.append(B[d])
            d += 1
    R.extend(A[g:])
    R.extend(B[d:])
    return R
```

$T(n)=3(n+1) + 2n + 4 + O(k) \ / \ k<n$

Donc : $T(n) = O(n)$

Complexité des fonctions récursives

```
def tri_fusion(L):
    n = len(L)
    if n <= 1:
        return L
    m=n//2
    G = tri _fusion(L[:m])
    D = tri _fusion(L[m:])
    return fusionner( G, D )
```

$T(n) = 2\,T(n/2) + 3n + 8$

Complexité quasi linéaire : $O(n \log(n))$

Tri par fusion

Avantages :

1. **Stabilité** : Le tri fusion est stable, ce qui signifie que l'ordre relatif des éléments égaux est préservé. Cela peut être essentiel lorsque vous triez des données par plusieurs critères.
2. **Performance garantie** : Le tri fusion garantit un temps d'exécution de O(n log n) dans le pire des cas. Cela en fait l'un des algorithmes de tri les plus efficaces pour trier de grandes quantités de données, en particulier lorsque la taille des données est importante.
3. **Adaptabilité** : Le tri fusion est bien adapté aux listes chaînées, car il peut être mis en œuvre de manière récursive, ce qui facilite le tri de structures de données liées.

Tri rapide

Principe :

L’algorithme de tri rapide, "quick sort", est un algorithme fondé sur la méthode de conception diviser pour régner; Son principe consiste à séparer l’ensemble des éléments en deux parties.

Pour effectuer la séparation, une valeur pivot est choisie au hasard. Les valeurs sont réparties en deux ensembles suivant qu’elles sont plus grandes ou plus petites que le pivot. Ensuite, les deux ensembles sont triés séparément, suivant la même méthode.

Tri Rapide – Quick Sort

1-Choisir un pivot

4	55	8	3	1	0	9	22	5	7

pivot

Tri Rapide – Quick Sort

2-Réorganiser les éléments autour du pivot

3	1	0	**4**	55	8	9	22	5	7

pivot

Tri Rapide – Quick Sort

3-Répétez 1 et 2

3	1	0	**4**	55	8	9	22	5	7

pivot pivot

Tri Rapide – Quick Sort

3-Répétez 1 et 2

1	0	**3**	**4**	8	9	22	5	7	**55**
		pivot							pivot

Tri Rapide – Quick Sort

3-Répétez 1 et 2

1	0	**3**	**4**	8	9	22	5	7	**55**

pivot pivot

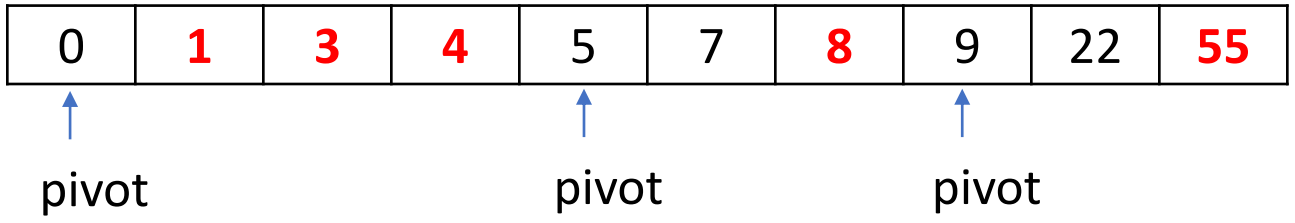
Tri Rapide – Quick Sort

3-Répétez 1 et 2

0	**1**	**3**	**4**	5	7	**8**	9	22	**55**
	pivot					pivot			

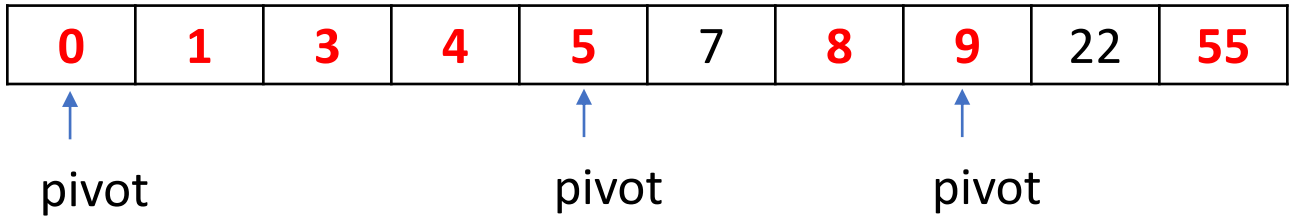
Tri Rapide – Quick Sort

3-Répétez 1 et 2



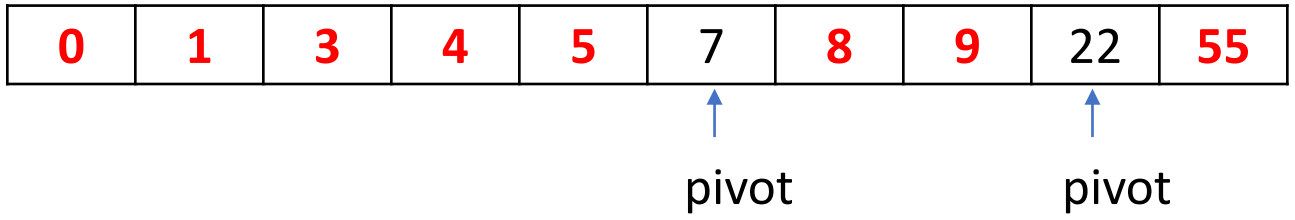
Tri Rapide – Quick Sort

3-Répétez 1 et 2



Tri Rapide – Quick Sort

3-Répétez 1 et 2



Tri Rapide – Quick Sort

3-Répétez 1 et 2

0	**1**	**3**	**4**	**5**	**7**	**8**	**9**	**22**	**55**

Tri rapide

Implémentation :

Ecrire la fonction « **partition(L, debut, fin)** » : qui prend en paramètre une liste L un indice de début et un indice de fin.

Cette fonction choisit un pivot et réorganise les éléments de la liste autour du pivot, puis renvoie le nouvel indice du pivot.

Ecrire la fonction « **tri_rapide(L, debut, fin)** » : : qui prend en paramètre une liste L un indice de début et un indice de fin. Cette fonction implémente l'algorithme du tri rapide, à la fin la liste L doit être triée.

Tri rapide

```
def partition(L, debut, fin):
    pivot = L[debut]
    ipivot = debut
    for i in range(debut+1, fin+1):
        if L[i]<=pivot :
            a = L.pop(i)
            L.insert(debut,a)
            ipivot = ipivot + 1
    return ipivot
```

Complexité de la fonction partition : O(n)

Tri rapide

```
def quick(L, debut, fin):
    if debut<fin :
        ipivot = partition(L, debut, fin)
        quick(L,debut,ipivot-1)
        quick(L,ipivot+1, fin)
```

$T(n) = 2 + O(n) + T(n-1) + O(1)$

$T(n) = T(n-1) + O(n)$

Complexité quadratique au pire : $O(n^2)$