Cours 4 : Tris de base

Passage en revue des algorithmes de tris les plus courants avec preuve et complexité.
Tableau récapitulatif

Algo	Complexité pire des cas	Complexité sur tableau trié	Complexité en moyenne
Tri sélection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Tri bulles	$O(n^2)$	$O(n)$	$O(n^2)$
Tri insertion	$O(n^2)$	$O(n)$	$O(n^2)$
Tri rapide	$O(n^2)$	$O(n^2)$	$O(n\log(n))$
Tri fusion	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$

1. Tri sélection

1.1. **Algorithme.** Ce tri est un des plus simples à implanter : le principe est de rechercher à chaque étape le maximum des données non triées et de le placer dans la position correcte.

```
TriSelection
Input :
    - tab un tableau de taille n
Procédé :
    Pour fin allant de n-1 à 1:
        imax <- 0
        Pour i allant de 1 à fin:
            Si tab[imax] <= tab[i]:
                imax <- i
        tab[imax], tab[fin] <- tab[fin], tab[imax]
```

On utilise la syntaxe python `u,v <- v,u` *pour échanger deux valeurs.*

Exemple 1.1.
Tri pas sélection du tableau

9 3 6 1 2 4 3 5 6 4

Pour chaque itération de la boucle sur la variable `fin`, on sélectionne le maximum puis on l'échange avec la dernière valeur.

`fin = 9`
9 3 6 1 2 4 3 5 6 4
4 3 6 1 2 4 3 5 6 9
`fin = 8`
La partie en bleu est triée
4 3 6 1 2 4 3 5 6 9
`fin = 7`
4 3 6 1 2 4 3 5 6 9
4 3 5 1 2 4 3 6 6 9
`fin = 6`
4 3 5 1 2 4 3 6 6 9
4 3 3 1 2 4 5 6 6 9
`fin = 5`
4 3 3 1 2 4 5 6 6 9
`fin = 4`
4 3 3 1 2 4 5 6 6 9
2 3 3 1 4 4 5 6 6 9
`fin = 3`
2 3 3 1 4 4 5 6 6 9
2 3 1 3 4 4 5 6 6 9
`fin = 2`
2 3 1 3 4 4 5 6 6 9
2 1 3 3 4 4 5 6 6 9
`fin = 1`
2 1 3 3 4 4 5 6 6 9
1 2 3 3 4 4 5 6 6 9

1.2. **Preuve.** La preuve d'un algorithme de ce type se fait en utilisant un *invariant de boucle*, c'est-à-dire une propriété qui restera vraie après chaque itération.

Invariant de boucle : A la fin de l'étape `fin`, on a conservé l'ensemble de valeurs initiales et les valeurs `tab[j]` pour $j \geq fin$ sont celles du tableau trié.

On prouve facilement par une logique de récurrence que cet invariant est vérifié à chaque étape. Supposons qu'il l'est pour `fin = k`, la valeur que l'on sélectionne à l'étape `fin = k-1` est le maximum des `tab[i]` pour $0 \leq i \leq k-1$ et par l'invariant de boucle, elle est plus petite ou égale que `tab[i]` pour $k \leq i \leq n-1$, donc elle correspond bien à la valeur finale du tableau trié en position $k-1$. Le fait qu'on effectue un échange assure la conservation des valeurs.

La boucle s'arrête quand `fin = 1` et donc toutes les valeurs `tab[i]` pour $1 \leq i \leq n-1$ sont bien placées, ce qui implique que `tab[0]` est lui aussi bien placé par conservation des valeurs initiales et que le tableau est trié.

1.3. **Complexité.** Le nombre d'itérations est constant quelles que soient les valeurs du tableau :

$$(n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}. \tag{1.1}$$

La complexité est donc en $O(n^2)$. Remarquez cependant que le nombre d'écriture dans le tableau est lui seulement en $O(n)$.

1.4. **Avantages / Inconvénients.** .

Avantages
- Implantation simple
- Faible nombre d'écritures
- Tri en place (directement dans le tableau sans allocation de mémoire supplémentaire)

Inconvénients
- Forte complexité quelle que soient les données (pas d'amélioration dans le meilleur des cas)

En pratique, ce tri est peu utilisé à cause de sa forte complexité *dans tous les cas.*

2. Tri à bulles

2.1. **Algorithme.** Le principe du tri est de comparer les données deux à deux sur des cases adjacentes et de continuer tant que l'on trouve des inversions. Pour qu'il soit optimal, il faut prendre avantage des propriétés des "bulles" qui poussent le maximum vers la fin du tableau ainsi que du test sur les inversions.

```
TriBulles
Input :
    - tab un tableau de taille n
Procédé :
    Pour fin allant de n-1 à 1 :
        inversions <- Faux
        Pour i allant de 0 à fin-1 :
            Si tab[i] > tab[i+1]:
                tab[i], tab[i+1] <- tab[i+1], tab[i]
                inversions <- Vrai
        Si inversions est faux :
            Quitter la boucle
```

Exemple 2.1.
Tri à bulles du tableau

9 3 6 1 2 4 3 5 6 4

9	**3**	6	1	2	4	3	5	6	4
3	**9**	**6**	1	2	4	3	5	6	4
3	6	**9**	**1**	2	4	3	5	6	4
3	6	1	**9**	**2**	4	3	5	6	4
3	6	1	2	**9**	**4**	3	5	6	4
3	6	1	2	4	**9**	**3**	5	6	4
3	6	1	2	4	3	**9**	**5**	6	4
3	6	1	2	4	3	5	**9**	**6**	4
3	6	1	2	4	3	5	6	**9**	**4**
3	**6**	1	2	4	3	5	6	4	9
3	**6**	**1**	2	4	3	5	6	4	9
3	1	**6**	**2**	4	3	5	6	4	9
3	1	2	**6**	**4**	3	5	6	4	9
3	1	2	4	**6**	**3**	5	6	4	9
3	1	2	4	3	**6**	**5**	6	4	9
3	1	2	4	3	5	**6**	**6**	4	9
3	1	2	4	3	5	6	**6**	**4**	9
3	**1**	2	4	3	5	6	4	6	9

1	**3**	**2**	4	3	5	6	4	6	9
1	2	**3**	**4**	3	5	6	4	6	9
1	2	3	**4**	**3**	5	6	4	6	9
1	2	3	3	**4**	**5**	6	4	6	9
1	2	3	3	4	**5**	**6**	4	6	9
1	2	3	3	4	5	**6**	**4**	6	9
1	**2**	3	3	4	5	4	6	6	9
1	**2**	**3**	3	4	5	4	6	6	9
1	2	**3**	**3**	4	5	4	6	6	9
1	2	3	**3**	**4**	5	4	6	6	9
1	2	3	3	**4**	**5**	4	6	6	9
1	2	3	3	4	**5**	**4**	6	6	9
1	**2**	3	3	4	4	5	6	6	9
1	**2**	**3**	3	4	4	5	6	6	9
1	2	**3**	**3**	4	4	5	6	6	9
1	2	3	**3**	**4**	4	5	6	6	9
1	2	3	3	**4**	**4**	5	6	6	9

2.2. **Preuve.** La preuve du programme s'appuie sur les deux stratégies mises en place : réduction de la partie non triée du tableau et tests des inversions. On peut utiliser le même invariant de boucle que celui du tri par sélection : il est facile de voir qu'à la fin de chaque étape, le maximum se retrouve toujours à la dernière place non triée. Par ailleurs, il faut justifier l'arrêt en cour de route par les inversions :

— une inversion de `tab` est un couple $i < j$ tel que `tab[i] > tab[j]` (i et j ne sont pas forcément consécutifs),

— un tableau non trié contient au moins une inversion sur deux indices consécutifs,

donc si je n'ai observé aucune inversion sur les "bulles" de la partie non triée, je peux arrêter l'algorithme.

2.3. **Complexité.** Réfléchissons d'abord à la **complexité du pire des cas**. Si on se contente d'observer l'imbrication des boucles for en ignorant le tests des inversions, on trouve une complexité en $O(n^2)$ de la même façon que pour le tri sélection. Cependant, on pourrait se demander si le tests des inversions permet d'améliorer ce résultat, en d'autre terme : est-ce que le pire des cas de la boucle for est atteint lorsqu'on ajoute le test des inversions ?

Pour répondre à cette question, il faut observer un peu plus finement le mécanisme. Soit I, le nombre d'inversions dans mon tableau et C le nombre de comparaisons effectuées par l'algorithme. Dans l'exemple ci-dessus, on trouve 20 inversions et 35 comparaisons.

— À chaque fois que l'on effectue un échange, on supprime exactement une inversion,

— il faut supprimer toutes les inversions pour obtenir le tableau trié,

— chaque échange suppose que l'on effectue une comparaison, donc on a $I \leq C$.

Et l'on sait par les bornes de la boucle for que $C \leq \frac{n(n-1)}{2}$. Soit un tableau de taille n, la configuration qui maximise le nombre d'inversion est l'ordre décroissant strict, dans ce cas on trouve $I = \frac{n(n-1)}{2} \leq C \leq \frac{n(n-1)}{2}$. Autrement dit, on atteint bien le nombre maximal de comparaisons : le tests des inversions n'améliore pas l'algorithme dans le pire des cas. **La complexité reste en $O(n^2)$.**

Cependant, il est évident que dans le **meilleur des cas** (tableau trié), le test des inversions permet d'obtenir une complexité $O(n)$. En pratique,on va souvent améliorer le nombre d'opérations comme on peut le voir dans l'exemple où l'algorithme se termine quand `fin = 5`.

Est-ce que cette amélioration se répercute sur la **complexité en moyenne** ? Pour répondre, il faut calculer le nombre d'inversions moyen d'un "tableau aléatoire". Pour simplifier, on supposera que toutes les valeurs sont distinctes et que le tableau correspond à une permutation aléatoire choisie uniformément parmi les permutations d'un certain ensemble.

Soit i et j, avec $i < j$, deux indices du tableau, la probabilité de $tab[i] < tab[j]$ est $\frac{1}{2}$ car pour deux valeurs v et w, le nombre de tableaux tel que $t[i] = v$ et $t[j] = w$ est le même que le nombre de tableau tel que $t[i] = w$ et $t[j] = w$. Il existe donc $\frac{n(n-1)}{2}$ couples $i < j$ et chacun a une probabilité $\frac{1}{2}$ d'être une inversion. Par linéarité de l'espérance, on obtient donc que le nombre d'inversions est en moyenne de $I = \frac{n(n-1)}{4}$, et comme on sait que $I \leq C$, on en conclue que la **complexité en moyenne est toujours de $O(n^2)$.**

2.4. **Avantages / Inconvénients.** .

Avantages
- Implantation simple
- Tri en place
- Complexité linéaire quand le tableau est trié

Inconvénients
- Forte complexité du pire des cas
- Forte complexité en moyenne
- Une seule valeur mal placée peut induire une complexité $O(n^2)$

3. Tri par insertion

3.1. **Algorithme.** Le principe du tri par insertion est de conserver triée la partie initiale du tableau et d'y insérer les éléments non triés un à un.

```
TriInsertion
Input :
    - tab un tableau de taille n
Procédé :
    Pour i allant de 1 à n-1 :
        j <- i-1
        v <- tab[i]
        Tant que j >= 0 et t[j] >v :
            t[j+1] <- t[j]
            j <- j - 1
        t[j+1] <- v
```

L'élément en position i est celui que l'on insère dans le tableau trié (éléments placés avant i). La boucle Tant que permet décaler les valeurs triées jusqu'à la position de la valeur à insérer.

Exemple 3.1.
Tri par insertion du tableau
9 3 6 1 2 4 3 5 6 4

9	**3**	6	1	2	4	3	5	6	4
9	**9**	6	1	2	4	3	5	6	4
3	**9**	6	1	2	4	3	5	6	4
3	9	**6**	1	2	4	3	5	6	4
3	**9**	**9**	1	2	4	3	5	6	4
3	**6**	**9**	1	2	4	3	5	6	4
3	6	9	**1**	2	4	3	5	6	4
3	6	**9**	**9**	2	4	3	5	6	4
3	**6**	**6**	**9**	2	4	3	5	6	4
3	**3**	**6**	**9**	2	4	3	5	6	4
1	**3**	**6**	**9**	2	4	3	5	6	4
1	3	6	9	**2**	4	3	5	6	4
1	3	6	**9**	**9**	4	3	5	6	4
1	3	**6**	**6**	**9**	4	3	5	6	4
1	**3**	**3**	**6**	**9**	4	3	5	6	4
1	**2**	**3**	**6**	**9**	4	3	5	6	4
1	2	3	6	9	**4**	3	5	6	4
1	2	3	6	**9**	**9**	3	5	6	4
1	2	3	**6**	**6**	**9**	3	5	6	4

1	2	3	**4**	**6**	**9**	3	5	6	4
1	2	3	4	6	9	**3**	5	6	4
1	2	3	4	6	**9**	**9**	5	6	4
1	2	3	4	**6**	**6**	**9**	5	6	4
1	2	3	**4**	**4**	**6**	**9**	5	6	4
1	2	3	**3**	**4**	**6**	**9**	5	6	4
1	2	3	3	4	6	9	**5**	6	4
1	2	3	3	4	6	**9**	**9**	6	4
1	2	3	3	4	**6**	**6**	**9**	6	4
1	2	3	3	4	**5**	**6**	**9**	6	4
1	2	3	3	4	5	6	9	**6**	4
1	2	3	3	4	5	6	**9**	**9**	4
1	2	3	3	4	5	6	**6**	**9**	4
1	2	3	3	4	5	6	6	9	**4**
1	2	3	3	4	5	6	6	**9**	**9**
1	2	3	3	4	5	6	**6**	**6**	**9**
1	2	3	3	4	5	**6**	**6**	**6**	**9**
1	2	3	3	4	**5**	**5**	**6**	**6**	**9**
1	2	3	3	4	**4**	**5**	**6**	**6**	**9**

3.2. **Preuve.** L'invariant de boucle est le suivant : à la fin de l'étape i, les valeurs du tableaux `tab[j]` avec $j \leq i$ sont triées et correspondent à une permutation des valeurs du tableau initial placées en position inférieures ou égales à i.

Lorsque l'on commence une nouvelle étape, on récupère la valeur `v= t[i]` à insérer. La boucle interne permet de déplacer d'un cran vers la droite toutes les valeurs supérieures à v et de placer v dans l'emplacement ainsi dégagé. L'invariant de boucle est donc bien respecté.

3.3. **Complexité.** Le pire des cas est obtenu sur un tableau strictement décroissant. Dans ce cas, à chaque insertion, il faut décaler l'ensemble des valeurs et on obtient une **complexité en** $O(n^2)$.

Dans le meilleur des cas, si le tableau est déjà trié, on effectuera un seul parcourt sans retour en arrière et on aura donc une **complexité en** $O(n)$.

Pour ce qui est de la complexité en moyenne, le problème est assez similaires à celui du tri à bulles. En effet, à chaque insertion, on supprime un nombre d'inversion exactement égal au décalage que l'on fait subir au tableau trié. Observez par exemple l'insertion du nombre 2 dans l'exemple donné : on décale le 3, le 6 et le 9 et on supprime les trois inversions de ces nombres avec le 2 lui-même (surtout, on en supprime pas plus). On en arrive donc à la même conclusion que pour le tri à bulle : en moyenne, il faudra au minimum $\frac{n(n-1)}{4}$ opérations, soit une **complexité en** $O(n^2)$.

Cependant, il est à noter que si l'on peut maîtriser le nombre d'inversions : par exemple en limitant la distance possible entre une valeur et sa position finale, on limite aussi à coup sûr le nombre d'opérations. En effet, en observant l'algorithme, on observe que le nombre de comparaisons est pratiquement égal au nombre d'inversions. Ainsi, si on exprime le problème, non plus seulement en fonction de n, mais en fonction de n et I où I est le nombre d'inversions on trouve une complexité en $O(n + I)$. En particulier, si on travail sur des données dont le nombre d'inversions est en $O(n)$, alors on retrouve une complexité linéaire.

3.4. **Avantages / Inconvénients.** .

Avantages
- Tri en place
- Complexité linéaire quand le tableau est trié
- Complexité linéaire quand le tableau est "presque" trié

Inconvénients
- Forte complexité du pire des cas
- Forte complexité en moyenne

4. Tri rapide / Quick sort

4.1. **Algorithme.** Le tri rapide suit un principe de *diviser pour régner*. A chaque étape, on choisit un pivot p (ici la première valeur) et effectue un parcourt du tableau tel que toutes les valeurs $v \leq p$ soit au début du tableau et les autres à la fin. On place v à la fin des petites valeurs, c'est-à-dire à son emplacement final dans le tableau trié. Puis on lance le tri récursivement sur les deux moitiés du tableau données par v.

```
TriRapide
Input :
    - tab un tableau de taille n
    - deb, l'indice de départ du tableau (0 au premier appel)
    - fin, le premier indice hors du tableau (n au premier appel)
Procédé :
    Si fin - deb <=1:
        # Le tableau contient au maximum 1 élément
        Retourner
    pivot <- tab[deb]

    # Première phase : organiser les valeurs par rapport au pivot
    # toutes les valeurs aux indices <= i doivent être <=  à pivot
    # toutes les valeurs aux indices >=j doivent être > à pivot
    i <- deb + 1
    j <- fin - 1
    Tant que i <= j :
        Tant que i <= j et tab[i] <= pivot :
            i <- i + 1
        Tant que i <= j et tab[j] > pivot :
            j <- j - 1
        # On a augmenté i et diminue j au maximum
        Si i < j :
            # on sait que la valeur en i est une "grande" valeur
            # et que celle en j est une "petite" valeur
            tab[i],tab[j] <- tab[j],tab[i]
            i <- i + 1
            j <- j - 1
    # On remet le pivot à sa place
    # j contient la dernière "petite" valeur
    tab[deb],tab[j] <- tab[j],tab[deb]

    # Deuxième phase : appels récursifs

    TriRapide(t, deb, i-1)
    TriRapide(t, j+1, fin)
```

Exemple 4.1.
Tri rapide du tableau

9 3 6 1 2 4 3 5 6 4

Le pivot est indiqué en rouge, les valeurs placées définitivement sont en bleu, et celles considérées par l'appel de fonction de courant sont en vert. On a affiché quelques étapes de

l'algorithme du placement de pivot : les valeurs identifiées comme "petites" sont en gras, les "grandes" sont en italiques.

9	3	6	1	2	4	3	5	6	4
9	**3**	**6**	**1**	**2**	**4**	**3**	**5**	**6**	**4**
4	3	6	1	2	4	3	5	6	9
4	**3**	**3**	1	2	4	*6*	*5*	*6*	9
4	**3**	**3**	**1**	**2**	**4**	*6*	*5*	*6*	9
4	3	3	1	2	4	6	5	6	9
4	**3**	**3**	**1**	**2**	4	6	5	6	9
2	3	3	1	4	4	6	5	6	9
2	**1**	3	*3*	4	4	6	5	6	9
2	**1**	*3*	*3*	4	4	6	5	6	9

1	2	3	3	4	4	6	5	6	9
1	2	**3**	3	4	4	6	5	6	9
1	2	**3**	**3**	4	4	6	5	6	9
1	2	3	3	4	4	6	5	6	9
1	2	3	3	4	4	**6**	5	6	9
1	2	3	3	4	4	**6**	**5**	**6**	9
1	2	3	3	4	4	**6**	5	6	9
1	2	3	3	4	4	**6**	**5**	6	9
1	2	3	3	4	4	5	6	6	9

4.2. **Preuve.** La preuve se fait par récurrence.

- Un tableau de taille inférieure ou égale à 1 est déjà trié : la fonction ne fait rien
- Supposons que `TriRapide` fonctionne pour les taille plus petites que n. Soit $p = tab[0]$ le pivot. La première partie de la fonction place les valeurs plus petites ou égales à p à gauche d'un indice i et les autres à droites. Cette partie là se prouve avec un invariant de boucle : à la fin de chaque étape, la distance entre i et j diminue d'au moins 1 et tout ce qui est à gauche de i est plus petit ou égal à p et tout ce qui est à droite de j est plus grand que p. A la fin de la boucle, on échange p avec la position juste à la gauche de i. De cette façon, les valeurs à gauche de p sont exactement les valeurs plus petites ou égales à p, donc p est bien placé dans le tableau.
- Par récurrence, les appels récursifs sur les deux parties du tableau, à gauche et à droite de p trient les sous-tableau car ils sont de taille strictement plus petite que n (étant donné qu'aucun des deux ne contient le pivot), et donc le tableau est trié.

4.3. **Complexité.** A chaque étape, le tableau est divisé en deux sous-tableaux. La taille de ces sous-tableaux n'est pas constante, elle dépend du pivot. Si l'on choisit un tableau strictement décroissant ou croissant, à chaque étape n, il sera décomposé en un tableau de taille O et un tableau de taille $n-1$. Soit f la complexité de mon tri, on obtient la formule récursive

$$f(1) = 1 \tag{4.1}$$

$$f(n) = n + f(n-1). \tag{4.2}$$

En dé-récursivant : f est la somme des entiers de 1 à n. On a une **complexité quadratique** soit $O(n^2)$.

Le cas où le **tableau est trié** dans un sens ou dans l'autre est donc **le pire des cas** pour le tri rapide. Quel est le meilleur des cas ? Celui où le tableau est divisé en deux sous-parties de tailles équivalentes. Prenons par exemple la fonction récursive suivante

$$f(n \leq 1) = 1 \tag{4.3}$$

$$f(n) = n + 2f\left(\frac{n}{2}\right) \tag{4.4}$$

$$= n + 2\left(\frac{n}{2} + 2\left(\frac{n}{4} + \dots\right)\right) \tag{4.5}$$

En développant, les fractions se simplifient et on obtient $f(n) = n(\log(n)+1)$, c'est-à-dire une complexité en $O(n\log(n))$. Donc la complexité du tri rapide est améliorée si on divise le tableau en parts égales.

On peut utiliser cette propriété pour le calcul de la complexité en moyenne. Sans entrer dans les détails du calcul, si le tableau est une permutation aléatoire d'un certain ensemble, le choix du pivot le divisera en sous-parties qui seront équilibrées en moyenne et elles-mêmes des permutations aléatoires. Par une étude probabiliste et des calculs un peu plus poussés, on obtient que **la complexité en moyenne est en** $O(n\log(n))$.

4.4. **Avantages / Inconvénients.** .

Avantages
- Tri en place
- Complexité $O(n\log(n))$ en moyenne
- Connu pour sa rapidité en pratique

Inconvénients
- Forte complexité du pire des cas
- Forte complexité dans le cas trié ou presque trié

Le quicksort est très utilisé en pratique car il permet d'obtenir de très bonnes performances. Cependant, il est le plus souvent couplé à un algorithme de *randomisation* ou autres euristiques particulières pour éviter de tomber dans le pire des cas. Par ailleurs, il est à éviter quand les données sont déjà **triées ou presque triées**, dans ce cas, sa complexité est très mauvaise alors qu'un tri par insertion sera fait en temps linéaire.

5. Tri Fusion

5.1. **Algorithme.** Le principe du tri fusion est basé sur l'algorithme qui permet de fusionner deux listes triées en $O(n)$ en choisissant en les minimums successifs des deux listes. Tout comme le tri rapide, c'est un **tri récursif** qui divise le problème en 2 sous problèmes plus petits.

```
Fusion
Fusion de deux listes triées
Input :
    - t1 un tableau trié de taille n
    - t2 un tableau trié de taille m
Output :
    un tableau trié contenant les valeurs de t1 et t2
Procédé :
    t3 <- tableau de taille n+m
    i <- 0 # indice pour parcourir t1
    j <- 0 # indice pour parcourir t2
    k <- 0 # indice pour parcourir t3
    Tant que i < n et j < m:
        Si t1[i] <= t2[j]: # si < : tri pas stable !
            t3[k] <- t1[i]
            i <- i+1
            k <- k+1
        Sinon :
            t3[k] <- t2[j]:
            j <- j+1
            k <- k+1
    # on a terminé un des deux tableaux, on recopie les valeurs restantes
    Tant que i < n:
        t3[k] <- t1[i]
        i <- i+1
        k <- k+1
    Tant que j < m:
        t3[k] <- t2[j]
        j <- j+1
        k <- k+1
    Retourner t3
```

```
TriFusion
Input :
    - tab un tableau de taille n
Output :
    Une copie du tableau trié
Procédé :
    Si n <= 1 :
        Retourner copie(t)
    m <- n/2
    t1 <- tab[:m] # sous tableau des indices 0 inclus à m exclus
    t2 <- tab[m:] # sous tableau des indices m inclus à n exclus
    t1 <- TriFusion(t1)
    t2 <- TriFusion(t2)
    Retourner Fusion(t1,t2)
```

Exemple 5.1.
Tri fusion du tableau
9 3 6 1 2 4 3 5 6 4
On affiche les appels récursifs à `TriFusion` et `Fusion`.

`TriFusion` de 9 3 6 1 2 4 3 5 6 4
`TriFusion` de 9 3 6 1 2
`TriFusion` de 9 3
`TriFusion` de 9
`TriFusion` de 3
`Fusion` de **9** et 3
3 **9**
`TriFusion` de 6 1 2
`TriFusion` de 6
`TriFusion` de 1 2
`TriFusion` de 1
`TriFusion` de 2
`Fusion` de **1** et 2
1 2
`Fusion` de **6** et 1 2
1 2 **6**
`Fusion` de **3** **9** et 1 2 6
1 2 **3** 6 **9**

`TriFusion` de 4 3 5 6 4
`TriFusion` de 4 3
`TriFusion` de 4
`TriFusion` de 3
`Fusion` de **4** et 3
3 **4**
`TriFusion` de 5 6 4
`TriFusion` de 5
`TriFusion` de 6 4
`TriFusion` de 6
`TriFusion` de 4
`Fusion` de **6** et 4
4 **6**
`Fusion` de **5** et 4 6
4 **5** 6
`Fusion` de **3** **4** et 4 5 6
3 **4** 4 5 6
`Fusion` de **1** **2** **3** **6** **9** et 3 4 4 5 6
1 **2** **3** 3 4 4 5 **6** 6 **9**

5.2. **Preuve.** Tout comme pour le tri rapide, la preuve se fait par récurrence de façon immédiate.

- Un tableau de taille inférieure ou égale à 1 est déjà trié : la fonction retourne une copie de ce tableau
- Supposons que `TriFusion` fonctionne pour les tailles plus petites que n. Par récurrence, les tableaux `t1` et `t2` sont donc des copies triées correspondant aux valeurs respectives de la première moitié de `tab` et de la dernière moitié de `tab`.
- Il reste à prouver que la fusion de `t1` et `t2` retourne bien le tableau trié contenant les valeurs des deux tableaux. Cela se fait par invariant de boucle : à chaque étape de la boucle `Tant que` on ajoute une valeur au tableau `t3` qui est le minimum des

valeurs restantes. Les deux boucles `Tant que` à la fin de la fonction assurent qu'on a bien recopié toutes les valeurs.

5.3. **Complexité.** Dans le cas du tri fusion, on découpe systématiquement le tableau en deux parts égales (contrairement au tri rapide, cela ne dépend plus des valeurs). A chaque étape, il faut effectuer une copie du tableau (pour créer les sous-tableaux) et la fusion, cela se fait en $O(n)$. On obtient donc une formule récursive pour la complexité du type de

$$f(n \leq 1) = 1 \tag{5.1}$$

$$f(n) = n + 2f\left(\frac{n}{2}\right) \tag{5.2}$$

(5.3)

et donc une complexité en $O(n\log(n))$ **dans tous les cas.**

Notons aussi dans l'implantation naïve que nous avons faite du tri, la **complexité en mémoire** est elle aussi en $O(n\log(n))$. Il est possible en améliorant l'algorithme de la réduire à $O(n)$ (en gérant de façon efficace une seule copie du tableau qu'on utilise pour les fusions). L'écriture d'un tri fusion **en place** et **efficace** est un un exercice difficile qui a fait l'objet de différentes recherches et propositions, on peut obtenir une complexité globale de $O(n\log(n)^2)$ mais on reste moins efficace que le tri originel.

5.4. **Avantages / Inconvénients.** .

Avantages

- Complexité $O(n\log(n))$ dans le pire des cas et en moyenne

Inconvénients

- Complexité $O(n\log(n))$ dans le meilleur des cas (donc pas linéaire)
- Pas de tri en place : "beaucoup" d'allocation mémoire.