Chapitre 2

Algorithmes de tri

2.1 Introduction

Les algorithmes de tri constituent l'un des piliers fondamentaux de l'informatique et jouent un rôle essentiel dans le traitement et la manipulation efficace des données. Le tri, qui consiste à réorganiser les éléments d'une collection dans un ordre particulier, est une opération fréquemment utilisée dans de nombreux domaines de l'informatique, tels que la recherche d'informations, l'analyse de données, les bases de données et bien d'autres encore. Dans ce chapitre, nous explorerons un éventail de techniques sophistiquées et efficaces pour organiser des ensembles de données de manière ordonnée. Nous aborderons des algorithmes de tri classiques, tels que le tri par sélection, le tri par insertion, le tri à bulles, ainsi que des méthodes plus avancées telles que le tri rapide (quicksort), le tri fusion (mergesort). Nous examinerons les différences entre les méthodes en termes d'efficacité et de complexité pour permettre aux lecteurs de faire des choix éclairés quant à la sélection de la meilleure méthode en fonction du contexte spécifique du problème à résoudre.

2.2 Présentation

Soit A un tableau de n éléments. Le tri consiste à réorganiser les éléments de A de sorte que les éléments de A soient ordonnés. L'ordre des éléments peut être croissant ou décroissant, alphabétique ou numérique, ou même personnalisé, tout dépend du contexte du problème à résoudre et de la nature des données à trier. Par exemple, dans le cas d'un tableau d'entiers, nous pouvons vouloir trier les éléments dans l'ordre croissant, tandis que dans le cas d'un tableau de chaînes de caractères, nous pouvons vouloir trier les éléments dans l'ordre alphabétique. Tous les algorithmes de tri peuvent s'adapter à n'importe quel type de données, à condition que l'ordre des éléments soit défini (les éléments doivent être comparables).

2.3 Tri à bulles

Le tri à bulles est l'un des algorithmes de tri les plus simples et les plus intuitifs. Il est basé sur l'idée de parcourir le tableau plusieurs fois, en comparant les éléments adjacents et en les échangeant s'ils ne sont pas dans l'ordre. L'algorithme s'arrête lorsque le tableau est parcouru sans qu'aucun échange ne soit effectué. Le tri à bulles est un algorithme de tri stable, ce qui signifie que l'ordre des éléments égaux n'est pas modifié. Le tri à bulles est un algorithme de tri en place, ce qui signifie qu'il ne nécessite pas de mémoire supplémentaire, car il effectue les comparaisons et les échanges directement dans le tableau d'entrée.

2.3.1 Exemple

Considérons le tableau $A = [5, 1, 4, 2, 8]$ à trier en ordre croissant. Le tri à bulles procède comme suit :

— **Premier passage :** on parcourt le tableau et on compare les éléments adjacents. Si un élément est plus grand que son successeur, on les échange. Notez ici que l'opération de comparaison est choisi en fonction de l'ordre de tri souhaité. Si on veut trier les éléments dans l'ordre croissant ou même trier un autre type de données avec un ordre plus complexe, seul le test d'ordre (l'opération de comparaison) doit être modifié, le reste de l'algorithme reste inchangé. Dans notre exemple, on compare 5 et 1, 5 est plus grand que 1, donc les éléments ne sont pas dans l'ordre, on va donc les échanger. Le tableau devient :

$$A = [1, 5, 4, 2, 8].$$

On compare ensuite 5 et 4, 5 est plus grand que 4, les éléments ne sont pas dans l'ordre, on les échange et le tableau devient :

$$A = [1, 4, 5, 2, 8].$$

La même chose se produit pour les éléments 5 et 2, on les échange et le tableau devient :

$$A = [1, 4, 2, 5, 8].$$

Pour les éléments 5 et 8, 5 est plus petit que 8, les éléments sont bien dans l'ordre, et donc on ne fait rien. Noter qu'à l'issue du premier passage, le plus grand élément du tableau est placé à la fin de ce dernier.Par conséquent, à chaque passage, on peut réduire la taille du tableau à parcourir de 1. C'est à dire qu'on peut parcourir les premiers $n - 1$ éléments au lieu de n pour le deuxième passage, les premiers $n - 2$

éléments pour le troisième passage, et ainsi de suite.

— **Deuxième passage :** on parcourt les $n-1$ premiers éléments du tableau et on compare les éléments adjacents. Si un élément est plus grand que son successeur, on les échange. Dans notre exemple, on compare 1 et 4, et les éléments ne sont pas échangés car il sont dans l'ordre. Puis on compare 4 et 2, 4 est plus grand que 2, on les échange et le tableau devient :

$$A = [1, 2, 4, 5, 8].$$

On compare ensuite 4 et 5, 4 est plus petit que 5, les éléments sont dans l'ordre et le tableau reste inchangé. Comme nous l'avons expliqué précédemment, on n'a pas besoin de comparer le dernier élément du tableau car il est déjà dans l'ordre. La même chose s'applique pour l'avant dernier élément, dans le prochain passage.

— **Troisième passage :** on parcourt les $n-2$ premiers éléments du tableau et on compare les éléments adjacents. Dans ce cas, nous allons observer que tous les $n-2$ éléments sont déjà dans l'ordre, car aucun échange n'a été effectué lors de ce passage. L'algorithme s'arrête donc ici.

2.3.2 Implémentation

Nous montrons ci-dessous une implémentation de l'algorithme de tri à bulles en C :

```
1  int bubble_sort(int *A, int n) {
2    int i, j, tmp;
3    for (i = 0; i < n - 1; i++) {
4      ordered = 1;
5      for (j = 0; j < n - i - 1; j++) {
6        if (A[j] > A[j + 1]) {
7          tmp = A[j];
8          A[j] = A[j + 1];
9          A[j + 1] = tmp;
10         ordered = 0;
11       }
12     }
13     if (ordered) {
14       return;
15     }
16   }
17 }
```

Algorithm 2.1 – Tri à bulles - version itérative

La version précédente est une implémentation itérative de l'algorithme de tri à bulles. Nous pouvons également implémenter cet algorithme de manière récursive, comme suit :

```
1  int bubble_sort_rec(int *A, int n) {
2    int i, tmp;
3    if (n != 1) {
4      ordered = 1;
5      for (i = 0; i < n - 1; i++) {
6        if (A[i] > A[i + 1]) {
7          tmp = A[i];
8          A[i] = A[i + 1];
9          A[i + 1] = tmp;
10         ordered = 0;
11       }
12     }
13     if (!ordered) {
14       bubble_sort_rec(A, n - 1);
15     }
16   }
17 }
```

Algorithm 2.2 – Tri à bulles - version récursive

2.3.3 Complexité

L'opération de base qui nous intéresse dans l'algorithme de tri à bulles est la comparaison entre deux éléments adjacents. L'algorithme comporte deux boucles imbriquées, la première boucle assure que l'algorithme effectue plusieurs passages sur le tableau (au plus $n-1$ passages), et la deuxième boucle assure que l'algorithme parcourt les éléments du tableau. Le nombre d'éléments parcourus dans la deuxième boucle dépend du passage courant, c'est à dire que dans le premier passage, on parcourt tous les éléments du tableau (on fait $n-1$ comparaisons), dans le deuxième passage, on parcourt les $n-1$ premiers éléments du tableau, et ainsi de suite. Sur chaque itération dans la deuxième boucle, une seule comparaison est effectuée. L'algorithme s'arrête lorsque tous les éléments sont dans l'ordre au cours du dernier passage. Mais dans le pire des cas (le tableau est trié dans l'ordre inverse), on effectue $n-1$ passages. Le nombre de comparaisons effectuées dans le pire des cas est donc :

$$T_{bubble}(n) = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}. \qquad (2.1)$$

L'algorithme a donc une complexité quadratique, c'est à dire $O(n^2)$.

On peut appliquer le même raisonnement pour la version récursive de l'algorithme, en remarquant que le nombre de comparaisons effectuées pour un tableau de taille n est $n-1$ comparaison. Puis on fait appel à la fonction récursivement avec un tableau de taille

$n-1$ et ainsi de suite. Le processus récursif s'arrête lorsque le tableau est trié, ou au pire, lorsque on fait un appel récursif avec un tableau de taille 1 (c'est à dire après $n-1$ appels récursifs).

2.4 Tri par sélection

Le tri par sélection est un algorithme de tri qui consiste à chercher le plus petit élément du tableau et à le placer à la première position (dans le cas d'un tri croissant), puis à chercher le deuxième plus petit élément et à le placer à la deuxième position, et ainsi de suite. L'algorithme de tri par sélection est aussi un algorithme en place, c'est à dire qu'il ne nécessite pas de mémoire supplémentaire pour effectuer le tri. L'algorithme s'arrête lorsque tous les éléments sont dans l'ordre.

2.4.1 Exemple

Nous allons illustrer le fonctionnement de l'algorithme de tri par sélection sur l'exemple suivant :

$$A = [5, 1, 4, 2, 8].$$

— Lors du premier passage, on cherche le plus petit élément du tableau, qui est 1, et on l'échange avec l'élément à la première position du tableau. Le tableau devient :

$$A = [1, 5, 4, 2, 8].$$

— Lors du deuxième passage, on cherche le plus petit élément du tableau, mais cette fois en ignorant le premier élément. Le plus petit élément à partir de la deuxième position est 2, on l'échange avec l'élément à la deuxième position du tableau. Le tableau devient :

$$A = [1, 2, 4, 5, 8].$$

Notez qu'à ce stage, le tableau est déjà trié, mais contrairemnt à l'algorithme de tri à bulles, l'algorithme de tri par sélection n'a pas de moyen de savoir si le tableau est trié ou non, il continue donc à effectuer les passages suivants, en éliminant les $k-1$ premiers éléments du tableau à chaque passage, où k est le numéro du passage courant. Le tableau va rester inchangé jusqu'au dernier passage.

2.4.2 Implémentation

Nous montrons ci-dessous une implémentation de la version itérative de l'algorithme de tri par sélection en langage C :

```
1  int selection_sort(int *A, int n) {
2    int i, j, ind_min, tmp;
3    for (i = 0; i < n - 1; i++) {
4      ind_min = i;
5      for (j = i + 1; j < n; j++) {
6        if (A[j] < A[ind_min]) {
7          ind_min = j;
8        }
9      }
10     if (ind_min != i) {
11       tmp = A[i];
12       A[i] = A[ind_min];
13       A[ind_min] = tmp;
14     }
15   }
16 }
```

Algorithm 2.3 – Tri par sélection - version itérative

Nous montrons ci-dessous une implémentation de la version récursive de l'algorithme de tri par sélection en langage C :

```
1  int selection_sort_rec(int *A, int n) {
2    int i, ind_min, tmp;
3    if (n != 1) {
4      ind_min = 0;
5      for (i = 1; i < n; i++) {
6        if (A[i] < A[ind_min]) {
7          ind_min = i;
8        }
9      }
10     if (ind_min != 0) {
11       tmp = A[0];
12       A[0] = A[ind_min];
13       A[ind_min] = tmp;
14     }
15     selection_sort_rec(A + 1, n - 1);
16   }
17 }
```

Algorithm 2.4 – Tri par sélection - version récursive

2.4.3 Complexité

L'opération de base à laquelle nous nous intéressons ici est aussi la comparaison. L'algorithme comporte deux boucles imbriquées, la première boucle assure que l'algorithme effectue $n-1$ passages sur le tableau, et la deuxième boucle parcourt les éléments du tableau à partir de la position $i+1$ (où i est le numéro du passage courant) pour chercher le minimum. Le nombre de comparaisons effectuées dans la deuxième boucle dépend du passage courant, c'est à dire que dans le premier passage, on parcourt $n-1$ éléments, dans le deuxième passage, on parcourt $n-2$ éléments, et ainsi de suite. Le nombre de comparaisons effectuées est donc :

$$T_{selection}(n) = (n-1) + (n-2) + \cdots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}. \tag{2.2}$$

L'algorithme a donc une complexité quadratique, c'est à dire $O(n^2)$.

On peut appliquer le même raisonnement pour la version récursive de l'algorithme d'une manière similaire à l'algorithme de tri à bulles. On arrive toujours à la même complexité quadratique.

2.4.4 Comparaison avec le tri à bulles

Lorsqu'on considère la comparaison comme l'opération de base, l'algorithme de tri par selection effectue toujours le même nombre de comparaisons indépendamment de l'ordre des éléments du tableau. Alors que l'algorithme de tri à bulles pourrais effectuer moins de comparaisons s'il trouve que le tableau est trié à la fin d'un passage. Néanmoins, l'algorithme de tri par sélection pourrais être plus efficace que l'algorithme de tri à bulles dans certains cas, car il effectue moins d'échanges que l'algorithme de tri à bulles.

Changement de perspective

Lorsqu'on considère l'échange comme l'opération de base, l'algorithme de tri par sélection un seul échange par passage au maximum. Vu que le nombre de passages est $n-1$, la complexité de l'algorithme de tri par sélection dans ce contexte est $O(n)$ (complexité linéaire). Alors que l'algorithme de tri à bulles effectue au pire un échange pour chaque comparaison (au pire des cas - le tableau est trié à l'envers), et le nombre de comparaisons est le même que le nombre d'échanges, la complexité de l'algorithme de tri à bulles dans ce contexte est $O(n^2)$ (complexité quadratique). Ce comportement explique la différence de performance entre les deux algorithmes.

— L'algorithme de tri par sélection est plus efficace que l'algorithme de tri à bulles s'il est exécuté sur une architecture matérielle où l'échange est plus coûteux que la

comparaison, surtout sur des processeurs ayant des registres de taille réduite ou une mémoire cache de petite taille.

— L'algorithme de tri à bulles est plus efficace que l'algorithme de tri par sélection s'il est exécuté sur une architecture matérielle où la comparaison est plus coûteuse que l'échange. L'opération de comparaison lorsqu'elle est utilisée dans une condition va engendrer un saut conditionnel, qui est une opération coûteuse en temps d'exécution, surtout sur des processeurs ayant des pipelines profonds ou un prédicteur de branchement peu performant.

La différence entre les deux algorithmes peut aussi dépendre de la nature des données à trier. Si les données sont déjà (presque) triées, l'algorithme de tri à bulles sera plus efficace que l'algorithme de tri par sélection, car il effectue moins de passages sur le tableau. L'algorithme à utiliser entre les deux dépend donc de la nature des données à trier et de l'architecture matérielle sur laquelle l'algorithme sera exécuté.

2.5 Tri par insertion

L'algorithme de tri par insertion est un algorithme de tri qui consiste à insérer chaque élément du tableau à sa place dans un tableau trié. L'algorithme commence par considérer que le premier élément du tableau est trié, puis il insère le deuxième élément à sa place dans le tableau trié, puis il insère le troisième élément à sa place dans le tableau trié, et ainsi de suite jusqu'à ce que tous les éléments du tableau soient insérés à leur place. L'algorithme de tri par insertion est un algorithme stable (l'ordre des éléments égaux est conservé) et en place (l'algorithme ne nécessite pas de mémoire supplémentaire). Cet algorithme se base sur la complexité de l'insertion dans un tableau trié, qui est sous linéaire (logarithmique) pour améliorer la complexité de l'algorithme de tri.

2.5.1 Exemple

Nous montrons ci-dessous un exemple d'exécution de l'algorithme de tri par insertion sur le tableau de 5 éléments suivant :

$$A = [\{5\}, 2, 4, 6, 1]$$

Le tableau A est divisé en deux parties, la première partie est le tableau trié, et la deuxième partie est le reste du tableau. La taille du tableau trié est initialement 1, et la taille du reste du tableau est initialement $n - 1$ (où n est la taille du tableau). L'algorithme commence par insérer le deuxième élément du tableau dans le tableau trié, puis il insère le troisième élément du tableau dans le tableau trié, et ainsi de suite jusqu'à ce que tous les éléments

du tableau soient insérés à leur place. Nous notons la partie triée du tableau en la mettant entre accolades.

— **Passage 1** : au premier passage, le tableau trié est $[\{5\}]$ et le reste du tableau est $[2, 4, 6, 1]$. La première étape consiste à insérer le deuxième élément du tableau (la valeur 2) dans le tableau trié, nous obtenons comme résultat :

$$A = [\{2, 5\}, 4, 6, 1]$$

.

— **Passage 2** : au deuxième passage, le tableau trié est $[\{2, 5\}]$ et le reste du tableau est $[4, 6, 1]$. Nous allons insérer la valeur 4 dans le tableau trié, nous obtenons comme résultat :

$$A = [\{2, 4, 5\}, 6, 1]$$

.

— **Passage 3** : au troisième passage, nous allons insérer la valeur 6 dans le tableau trié, nous obtenons comme résultat :

$$A = [\{2, 4, 5, 6\}, 1]$$

.

— **Passage 4** : le quatrième passage consiste à insérer le dernier élément du tableau (la valeur 1) dans la partie triée du tableau, ce qui nous donne le tableau trié suivant :

$$A = [\{1, 2, 4, 5, 6\}]$$

.

2.5.2 implémentation

Avant d'implémnter l'algorithme de tri par insertion, il est très important d'implémenter et comprendre le processus d'insertion dans un tableau trié. La version naive de l'insertion dans un tableau trié consiste à parcourir le tableau à la recherche de la position d'insertion (le premier élément qui est plus grand que l'élément à insérer dans le cas d'un tri croissant), puis décaler tous les éléments du tableau à partir de la position d'insertion d'une case vers la droite, puis insérer l'élément à sa place. Cette version de l'insertion est implémentée en langage C ci-dessous :

Insertion dans un tableau trié

```
1  void insert(int *array, int size, int element) {
2    int i = 0;
3    while (i < size && array[i] < element) i++;
4    for (int j = size; j > i; j--) {
5      array[j] = array[j - 1];
6    }
7    array[i] = element;
8  }
```

Algorithm 2.5 – Insertion dans un tableau trié - Complexité linéaire

Noter que la fonction `insert` prend en paramètre un tableau d'entiers `array`, la taille du tableau `size` et l'élément à insérer `element`. La fonction `insert` retourne le tableau `array` avec l'élément `element` inséré à sa place en supposant que le tableau `array` est suivi d'une case vide utilisable pour assurer que le décalage des éléments du tableau ne dépasse pas sa taille.

Il est possible d'implémenter une version plus efficace de l'insertion dans un tableau trié en utilisant la recherche dichotomique pour trouver la position d'insertion de l'élément. La recherche dichotomique est une recherche qui consiste à diviser le tableau en deux parties, puis à chercher dans quelle partie se trouve l'élément à chercher, le tableau est ensuite divisé en deux parties à chaque fois de la façon suivante :

— Si l'élément à insérer est plus grand que l'élément au milieu du tableau, alors l'élément doit être inséré dans la deuxième partie du tableau.

— Si l'élément à insérer est plus petit que l'élément au milieu du tableau, alors l'élément doit être inséré dans la première partie du tableau.

Le même processus est répété jusqu'à ce que la position d'insertion de l'élément soit trouvée, ou on fini avec un tableau de taille 0.

La version de l'insertion dans un tableau trié utilisant la recherche dichotomique est implémentée en langage C ci-dessous :

```
1  void insert(int *array, int size, int element) {
2    int i = 0, j = size - 1, k;
3    while (i <= j) {
4      k = (i + j) / 2;
5      if (array[k] < element) i = k + 1;
6      else j = k - 1;
7    }
8    for (k = size - 1; k > i; k--) {
9      array[k] = array[k - 1];
```

```
10    }
11    array[i] = element;
12  }
```

Algorithm 2.6 – Insertion dans un tableau trié - Complexité logarithmique

Le processus de recherche de position d'insertion peut également être implémenté de façon récursive, ce qui donne la version suivante :

```
1   int search(int *array, int i, int j, int element) {
2     if (i > j) return i;
3     int k = (i + j) / 2;
4     if (array[k] < element) return search(array, k + 1, j, element);
5     else return search(array, i, k - 1, element);
6   }
7
8   void insert(int *array, int size, int element) {
9     int i = search(array, 0, size - 1, element);
10    for (int k = size - 1; k > i; k--) {
11      array[k] = array[k - 1];
12    }
13    array[i] = element;
14  }
```

Algorithm 2.7 – Insertion dans un tableau trié - Version récursive

Une fois que nous avons implémenté la fonction d'insertion dans un tableau trié, nous pouvons maintenant passer au tri par insertion. Nous allons parcourir le tableau à trier en insérant chaque élément dans le tableau trié, ce qui nous donne l'implémentation suivante :

```
1   void insertion_sort(int *array, int size) {
2     for (int i = 1; i < size; i++)
3       insert(array, i, array[i]);
4   }
```

Algorithm 2.8 – Tri par insertion

2.5.3 Complexité

Nous considérons la comparaison comme l'opération de base de l'algorithme de tri par insertion. Sa complexité dépend principalement de la complexité de la fonction d'insertion dans un tableau trié. Dans le cas ou on utilise la recherche dichotomique pour trouver la position d'insertion de l'élément, la complexité de la fonction d'insertion est logarithmique, vu que la recherche dichotomique dichotomique divise le tableau en deux parties à chaque fois. En d'autres termes, le nombre de comparaisons effectuées pour trouver la

position d'insertion dans un tableau trié de taille n est limité par le nombre de fois qu'on peut diviser n par 2, ce qui donne la complexité logarithmique au pire des cas $O(\log_2 n)$.

L'insertion est effectuée n fois dans le cas du tri par insertion, ce qui donne une complexité de $O(n \log n)$ au pire des cas. Cette complexité est un peu pessimiste, car la taille du tableau dans lequel on insère les éléments commence à 1 et augmente à chaque fois, ce qui fait que le nombre de comparaisons effectuées pour insérer les éléments est donné par la formule suivante :

$$T_{ins_sort}(n) = \log_2 1 + \log_2 2 + \log_2 3 + \cdots + \log_2 n \tag{2.3}$$

Mais le resultat de cette somme est toujours inférieur à la somme suivante :

$$\log_2 n + \log_2 n + \log_2 n + \cdots + \log_2 n = n \log_2 n \tag{2.4}$$

D'ou la complexité $O(n \log_2 n)$.

2.5.4 Comparaison avec les autres algorithmes de tri

La comparaison entre les algorithmes de tri est une tâche difficile, car plusieurs opérations peuvent être considérées comme des opérations de base. Très souvent, on considère le nombre de comparaisons comme l'opération de base, et dans ce cas, il semble que le tri par insertion est plus efficace que le tri par sélection et le tri à bulle. Cependant, le tri par insertion effectue plus d'opéarations mémoires que le tri par sélection. En effet, au pire des cas, l'orsque les opérations mémoires sont considérées comme opérations de base, le tri par insertion a une complexité de $O(n^2)$, ce qui est la même complexité que le tri à bulle, alors que le tri par sélection a une complexité de $O(n)$, ce qui est plus efficace que les deux autres algorithmes. En plus de cela, l'algorithme de tri par insertion est adaptatif, c'est à dire que si le tableau est presque trié, alors le nombre d'opérations effectuées est très petit.

En conclusion, l'algorithme le plus efficace parmie les trois vus jusqu'ici dépend de la situation, les questions qu'il faut se poser sont :

— Est-ce que le tableau est partiellement trié ?

— Est-ce qu'une opération mémoire est plus coûteuse qu'une opération de comparaison ?

— Est-ce que le prédicteur de branchement est efficace ?

— Le nombre de registres est-il limité ?

— La mémoire cache est-elle suffisante ?

2.6 Tri par fusion

Le tri par fusion est un algorithme récursif qui divise le tableau à trier en deux parties, trie chaque partie, puis les fusionne pour obtenir le tableau trié. La fonction de fusion prend deux tableaux triés et les fusionne en un seul tableau toujours trié. L'algorithme de tri par fusion est un algorithme de tri externe, c'est à dire qu'il doit utiliser un espace mémoire externe pour stocker les données à trier. Les appels récursifs s'arrêtent lorsque le tableau à trier ne contient qu'un seul élément, car un tableau d'un seul élément est toujours trié.

Le tri par fusion est un algorithme de tri un peu avancé et il est très efficace lorsqu'il s'agit de trier un tableau de taille importante. Il est également très efficace lorsqu'il s'agit de trier des données stockées sur un support externe, car il recopie à chaque fois une partie des données à trier dans la mémoire principale, ce qui permet d'éviter les accès répétitifs à des mémoires externes qui sont très lents.

2.6.1 Exemple

Nous allons illustrer le fonctionnement de l'algorithme de tri par fusion en utilisant un exemple. Nous allons trier le tableau $A = [5, 3, 8, 6, 2, 7, 1, 9]$. Le processus de tri par fusion est illustré par la figure 2.1. Nous avons spécifiqement choisi un tableau de taille 2^k pour faciliter l'explication. Néanmoins, l'algorithme de tri par fusion fonctionne aussi pour les tableaux de taille non puissance de 2. Dans la partie supérieure de la figure 2.1, le tableau à trier est divisé en deux parties égales à chaque fois jusqu'à obtenir des tableaux de taille 1 (qui est trié par définition). Dans la partie inférieure, les tableaux triés sont fusionnés jusqu'à obtenir le tableau trié final.

2.6.2 Implémentation

L'algorithme de tri par fusion est implémenté en utilisant une fonction récursive qui prend en paramètre le tableau à trier avec sa taille. La fonction récursive divise le tableau en deux parties, puis appelle récursivement la fonction de tri sur chaque partie. Lorsque la taille du tableau est égale à 1, la fonction retourne le tableau. Lorsque les deux tableaux triés sont retournés, une autre fonction est appelée pour les fusionner en un seul tableau toujours trié. Nous allons commencer d'abord par l'implémentation de la fonction de fusion qui prend en paramètre deux sous tableaux triés successifs et les fusionne en un seul tableau trié qui occupe la même place que les deux tableaux fusionnés. La fonction doit utiliser un espace mémoire externe pour stocker le tableau fusionné, puis le copier dans le tableau

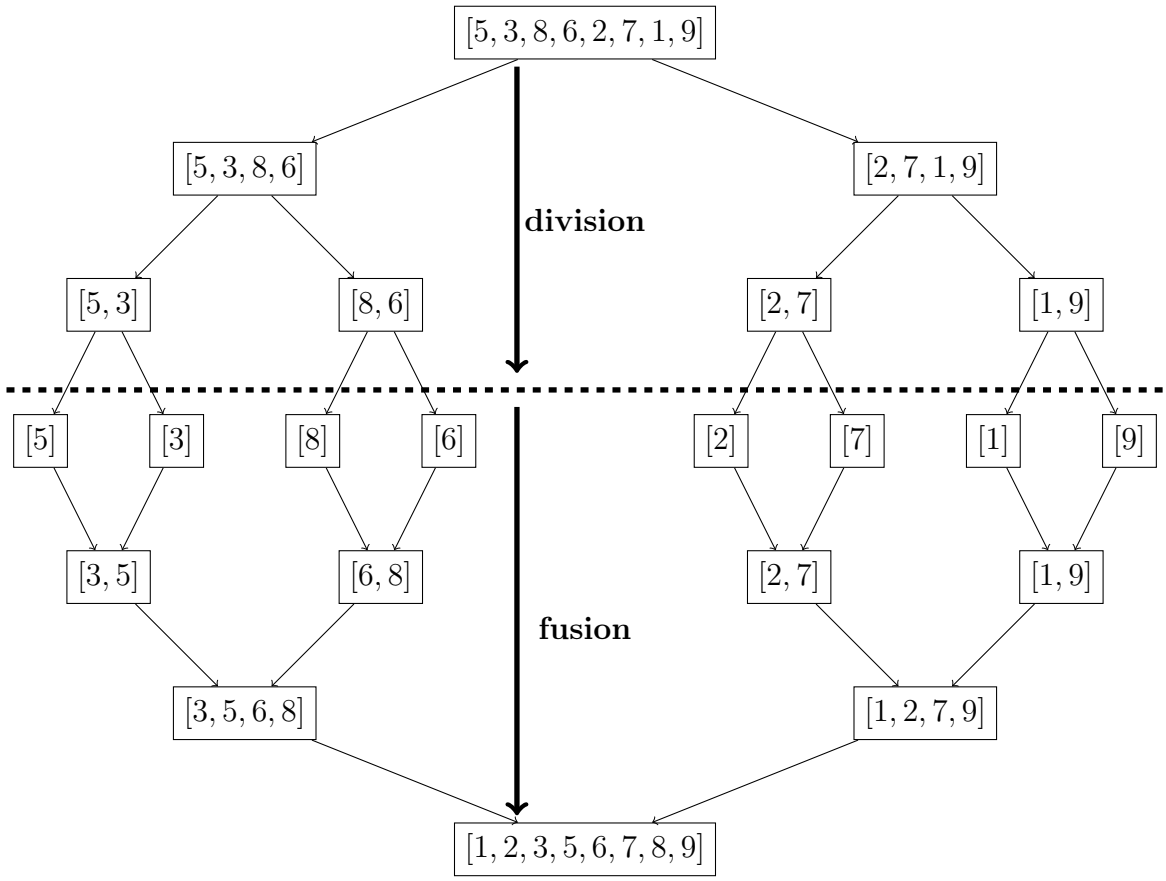

FIGURE 2.1 – Tri par fusion

initial. L'implémentation de la fonction de fusion est illustrée ci-dessous.

```
1  void merge(int *first, int *second, int size_first, int size_second) {
2    int *merged = malloc((size_first + size_second) * sizeof(int));
3    int i = 0, j = 0, k = 0;
4    while (i < size_first && j < size_second) {
5      if (first[i] < second[j]) {
6        merged[k] = first[i];
7        i++;
8      } else {
9        merged[k] = second[j];
10       j++;
11     }
12     k++;
13   }
14   while (i < size_first) {
15     merged[k] = first[i];
16     i++;
17     k++;
18   }
```

```
19    while (j < size_second) {
20      merged[k] = second[j];
21      j++;
22      k++;
23    }
24    for (i = 0; i < size_first + size_second; i++) {
25      first[i] = merged[i];
26    }
27    free(merged);
28  }
```

Algorithm 2.9 – Fusionner deux tableaux triés

Nous allons maintenant implémenter la fonction de tri par fusion. Cette fonction récursive prend en paramètre le tableau à trier et sa taille. Si la taille du tableau est égale à 1, alors la fonction retourne le même tableau. Sinon, la fonction divise le tableau en deux parties, puis appelle récursivement la fonction de tri sur chaque partie. Lorsque les deux tableaux triés sont retournés, on utilise la fonction de fusion dans l'algorithme 2.9 pour fusionner les fusionner. L'implémentation de la fonction de tri par fusion est illustrée ci-dessous :

```
1   void merge_sort(int *array, int size) {
2     if (size != 1) {
3       int size_first = size / 2;
4       int size_second = size - size_first;
5       int *first = array;
6       int *second = array + size_first;
7       merge_sort(first, size_first);
8       merge_sort(second, size_second);
9       merge(first, second, size_first, size_second);
10    }
11  }
```

Algorithm 2.10 – Tri par fusion

2.6.3 Complexité

Nous considérons comme pour les autres algorithmes que l'opération de comparaison est l'opération la plus coûteuse. Nous allons donc estimer la complexité de l'algorithme de tri par fusion en comptant le nombre de comparaisons effectuées. La complexité de l'algorithme de tri par fusion dépend de (1) la complexité de la fonction de fusion et (2) le nombre d'appels récursifs, c'est à dire la profondeur de la récursion. La complexité de la fonction de fusion est linéaire, car elle parcourt les deux tableaux à fusionner une seule fois. Le nombre d'appels récursifs ne peut pas dépasser le nombre de fois où on peut diviser la taille du tableau par 2 jusqu'à obtenir 1, ce qui est égal à $log_2 n$ où n est la taille du tableau initial. Vu qu'à chaque niveau de la récursion, chaque élément est comparé une

fois au maximum (n comparaisons à chaque niveau), et que la profondeur de la récursion est $log_2 n$, la complexité de l'algorithme de tri par fusion est de l'ordre de $O(n \cdot log_2 n)$.

Si on s'intéresse aux opérations mémoires, l'algorithme de tri par fusion a toujours une complexité de l'ordre de $O(n \cdot log_2 n)$, car à chaque niveau de la récursion, n éléments sont copiés dans un tableau externe, puis recopiés dans le tableau initial ($T(n) = 2n$). La profondeur de la récursion est $log_2 n$, donc la complexité de l'algorithme de tri par fusion est de l'ordre de $O(n \cdot log_2 n)$.

2.6.4 Comparaison avec les autres algorithmes

L'algorithme de tri par fusion est plus efficace que les autres algorithmes de tri que nous avons vus jusqu'à présent. En effet, la complexité de l'algorithme de tri par fusion est de l'ordre de $O(n \cdot log_2 n)$, quelque soit l'opération considérée. C'est la complexité optimale pour les algorithmes de tri par comparaison. L'algorithme de tri par fusion est donc plus efficace que les algorithmes de tri à bulle, par insertion et par sélection, particulièrement lorsque la taille du tableau à trier est grande.

2.7 Tri rapide (Quick sort)

Le tri rapide est un algorithme de tri par comparaison qui utilise la technique de **diviser et conquérir**. Comme le tri par fusion, il est basé sur la technique de **partitionnement** qui consiste à diviser le tableau en deux parties, puis à trier chaque partie séparément. L'algorithme de tri rapide est un algorithme récursif qui utilise une fonction de partitionnement pour diviser le tableau en deux parties. La fonction de partitionnement choisit un élément du tableau appelé **pivot**, puis place tous les éléments plus petits que le pivot à sa gauche et tous les éléments plus grands que le pivot à sa droite. L'algorithme de tri rapide est aussi un algorithme en place, car il ne nécessite pas de tableau externe pour effectuer le tri.

2.7.1 Exemple

Nous allons illustrer le fonctionnement de l'algorithme de tri rapide sur le tableau suivant :

$$A = [5, 3, 7, 2, 1, 4, 6]$$

Nous allons d'abord choisir un pivot. Ce dernier peut être choisi de différentes manières, mais le choix le plus courant est de choisir le dernier élément du tableau. Dans notre

exemple, le pivot est égal à 6. Nous allons ensuite parcourir le tableau de gauche à droite, et placer tous les éléments plus petits que le pivot à sa gauche. Lorsqu'on trouve un élément plus petit que le pivot, on l'échange avec le premier élément plus grand que le pivot. Lorsqu'on a parcouru tout le tableau, on échange le pivot avec le premier élément plus grand que lui. Le tableau devient alors :

$$A = [\{5, 3, 2, 1, 4\}, \mathbf{6}, \{7\}]$$

Par la suite, on applique la même procédure sur les deux parties du tableau séparées par le pivot. La partie droite du tableau est déjà triée, car elle ne contient qu'un seul élément. On applique donc la procédure sur la partie gauche du tableau. Le dernier élément du sous tableau est choisi comme pivot, sa valeur est 4, après cette étape, le tableau devient :

$$A = [\{3, 2, 1\}, \mathbf{4}, \{5\}, 6, 7]$$

On applique la même procédure sur le sous tableau de $[3, 2, 1]$, le dernier élément est choisi comme pivot, sa valeur est 1, après cette étape, le tableau devient :

$$A = [\mathbf{1}, \{2, 3\}, 4, 5, 6, 7]$$

À la dernière étape, on applique la même procédure sur le sous tableau de $[2, 3]$, il est déjà trié, mais pour finir l'exemple on va quand même choisir le dernier élément comme pivot, sa valeur est 3, après cette étape, le tableau devient :

$$A = [1, \{2\}, \mathbf{3}, 4, 5, 6, 7]$$

À l'issue de cette dernière étape, le tableau est trié.

2.7.2 Implémentation

L'implémentation de l'algorithme de tri rapide est donnée ci-dessous :

```
1  void quick_sort(int *array, int size) {
2    if (size > 1) {
3      int pivot = array[size - 1];
4      int i = 0;
5      int j = size - 2;
6      while (i <= j) {
7        if (array[i] > pivot && array[j] < pivot) {
8          int tmp = array[i];
```

```
 9           array[i] = array[j];
10           array[j] = tmp;
11           i++;
12           j--;
13         } else if (array[i] <= pivot) {
14           i++;
15         } else if (array[j] >= pivot) {
16           j--;
17         }
18       }
19       int tmp = array[i];
20       array[i] = array[size - 1];
21       array[size - 1] = tmp;
22       quick_sort(array, i);
23       quick_sort(array + i + 1, size - i - 1);
24     }
25   }
```

2.7.3 Complexité

Considérons l'opération de comparaison comme l'opération de base. La complexité de l'algorithme de tri rapide au pire des cas (lorsque le tableau est déjà trié) est de l'ordre de $O(n^2)$, car à chaque niveau de la récursion, chaque élément est comparé au pivot une fois au maximum ($n - i$ comparaisons tel que i est le niveau de la récursion). La profondeur de la récursion est n, donc la complexité de l'algorithme de tri rapide est de l'ordre de $O(n^2)$. La complexité de l'algorithme de tri rapide au meilleur des cas (lorsque le pivot est toujours au milieu du tableau) est de l'ordre de $\Omega(n \cdot log_2 n)$, car à chaque niveau de la récursion, chaque élément est comparé au pivot une fois au maximum ($n-i$ comparaisons tel que i est le niveau de la récursion). La profondeur de la récursion est $log_2 n$, donc la complexité au meilleur cas de l'algorithme de tri rapide est de l'ordre de $\Omega(n \cdot log_2 n)$. La complexité de l'algorithme de tri rapide en moyenne est de l'ordre de $n \cdot log_2 n$, car sur des tableaux aléatoires de taille n, il est très peu probable d'avoir un tableau déjà trié.

2.7.4 Comparaison avec le tri par fusion

Bien que la complexité au pire de l'algorithme de tri rapide soit de l'ordre de $O(n^2)$ dans le pire des cas, cet algorithme est en moyenne plus efficace que l'algorithme de tri par fusion. En effet, la complexité moyenne de l'algorithme de tri rapide est de l'ordre de $O(n \cdot log_2 n)$, la même que celle de l'algorithme de tri par fusion. De plus, l'algorithme de tri rapide est un algorithme en place, alors que l'algorithme de tri par fusion nécessite un tableau externe pour effectuer le tri. En plus, dans le pire des cas pour l'algorithme de tri rapide, aucune opération de copie n'est effectuée, et la comparaison avec le pivot donne

toujours un résultat négatif, ce qui permet au prédicteur de branchement de s'adapter et d'optimiser les performances de l'algorithme.

2.8 Conclusion

Dans ce chapitre, nous avons présenté plusieurs algorithmes de tri. Nous avons commencé par les algorithmes les plus simples à comprendre et à implémenter, le tri à bulles et le tri par selection. Ces deux algorithmes ont une complexité de l'ordre de $O(n^2)$, et sont donc très peu efficaces sur des tableaux de grande taille. Nous avons ensuite présenté le tri par insertion, qui a une complexité de l'ordre de $O(n \cdot log_2 n)$ lorsqu'il s'agit de comparaisons, mais de l'ordre de $O(n^2)$ si nous considérons les opérations mémoires. Nous avons ensuite présenté le tri par fusion, qui a une complexité de l'ordre de $O(n \cdot log_2 n)$, qui est beaucoup plus rapide que les algorithmes précédents sur des tableaux de grande taille. Finalement, l'algorithmes de tri rapide a été présenté, il a une complexité au pire de l'ordre de $O(n^2)$, mais en pratique il est plus rapide que le tri par fusion, car il est en place et que ça complexité moyenne est de l'ordre de $n \cdot log_2 n$.

Le choix de l'algorithme de tri à utiliser dépend de plusieurs facteurs, comme la taille du tableau à trier, la complexité de l'algorithme, la nature des données à trier, la quantité de mémoire disponible, etc. En pratique, le tri rapide est l'algorithme de tri le plus utilisé, y compris dans les bibliothèques standard des langages de programmation. Cependant, dans certaines situations, d'autres algorithmes de tri peuvent être plus efficaces.