# Chapitre 5

# Algorithmes de Recherche et de Tri

*Ce chapitre présente plusieurs algorithmes classiques de recherche et de tri.*

## 1- Problèmes de recherche et de tri

La majorité des applications informatiques ont besoin de stocker des données en grande quantité. Avant de traiter une donnée, il faut savoir la rechercher pour la retrouver rapidement. Le tri des données est une solution fondamentale pour accélérer la recherche d'une donnée particulière dans un domaine vaste.

En général, le critère de recherche d'une donnée porte sur la valeur d'une **clé**. Par exemple, les banques trient les chèques des clients par leurs numéros.

Les algorithmes de recherche peuvent varier selon plusieurs critères :

- S'agit-il d'une **recherche interne** (dans la mémoire centrale RAM) ou d'une **recherche externe** (sur le disque dur de l'ordinateur).
- La **clé de la recherche** est-elle **simple** (un numéro) ou **composée** (un nom + un prénom + une date de naissance) ?
- La **structure de données utilisée** : tableau, liste linéaire, liste doublement chaînée, …etc.
- La **nature de la recherche** : est-ce qu'on recherche une occurrence, la première occurrence, toutes les occurrences des éléments ayant la même clé, …etc.

### Définition formelle du problème de recherche étudié :

**Problème** : étant donné un tableau $A$ de $N$ éléments (des entiers, par exemple) et un élément $e$ (à rechercher). Trouver un indice d'un élément quelconque $x$ du tableau $A[1..N]$ ayant la même valeur que l'élément $e$ (c'est-à-dire, tel que $x = E$).

**Entrée** : un entier $N$, un tableau $A[1..N]$ de $N$ éléments entiers et un élément $e$ (un entier).

**Sortie** : un indice (un entier) d'un élément $x$ dans le tableau $A[1…N]$ tel que $x = e$, ou bien 0 si l'élément $e$ ne figure pas parmi les éléments du tableau A[1…$N$].

### Définition formelle du problème de tri étudié :

**Problème** : étant donné un tableau $A$ de $N$ éléments (des entiers, par exemple). Trier (ou ordonner) le tableau $A[1..N]$ dans l'ordre croissant.

**Entrée** : un entier $N$, une liste de $N$ éléments (entiers) : $\langle a_1, a_2, \ldots, a_N \rangle$.

**Sortie** : une permutation (réarrangement) $\langle a'_1, a'_2, \ldots, a'_N \rangle$ de la liste d'entrée telle que $a'_1 \leq a'_2 \leq \ldots \leq a'_N$.

# 2- Algorithmes de recherche

## A) Algorithme de recherche séquentielle :

**Principe** :

- Parcourir le tableau $A$ du début jusqu'à ce qu'on trouve un élément $x = e$.
- Lorsqu'on visite un élément $x$ d'indice $i$, on le compare avec $e$. Si $x = e$, on s'arrête et on retourne $i$, sinon on continue le parcours du tableau (on incrémente la valeur de $i$).

**Algorithme** :

**Fonction** RechSéq($A$ : **Tableau d'Entier**, $n$, $e$ : **Entier**) : **Entier**

**Variables** $i$ : **Entier**;

**Début**

$i$ := 1;

**Tant Que** ($i \leq n$)

**Si** ($A[i] = e$) **Alors** **Retourner**($i$);

**Sinon** $i$ := $i + 1$;

**FinSi**

**FinTQ**

**Retourner**(0);

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

- Dans le meilleur des cas :
  - Configuration qui donne le meilleur des cas : $A[1] = e$.
  - $Min_{RechSéq}(n) = 1$.
- Dans le pire des cas :
  - Configuration qui donne le pire des cas : $e \notin A$, ou bien $A[n] = e$ avec $A[i] \neq e, \forall i < n$.
  - $Max_{RechSéq}(n) = n$.
- L'algorithme de recherche séquentiel a donc une complexité en O($n$) (algorithme linéaire).

## B) Algorithme de recherche dichotomique :

**Principe** :

Lorsque le tableau initial est déjà trié, on peut améliorer la recherche en procédant, par une stratégie de type diviser pour régner, comme suit :

- On compare l'élément recherché $e$ avec l'élément $m$ qui se trouve au milieu du tableau initial $A$ (dont l'indice est $j$).
- Si $m = e$, on retourne l'indice du milieu du tableau (c-à-d. $j$).
- Si $e < m$, on refait le même processus dans le tableau $A[1 \ldots j - 1]$.
- Si $e > m$, on refait le même processus dans le tableau $A[j + 1 \ldots n]$.

**Algorithme** :

**Fonction** RechDicho($A$ : **Tableau d'Entier**, $deb$, $fin$ : **Entier**) : **Entier**

**Variables** $m$ : **Entier**; *// indice du minimum*

**Début**

**Si** ($deb > fin$) **Alors** **Retourner**(0);

**Sinon** $m := \left\lfloor \frac{deb + fin}{2} \right\rfloor$;

**Si** ($A[m] = e$) **Alors** **Retourner**($m$);

**Sinon Si** ($A[m] > e$) **Alors**

**Retourner**(RechDicho(A, $deb$, $m$ - 1));

**Sinon**

**Retourner**(RechDicho(A, $m$ + 1, $fin$));

**FinSi**

**FinSi**

**FinSi**

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

Calculons la complexité de cet algorithme dans le pire des cas :

- Soit $n = fin - deb + 1$ la taille du tableau (c'est la taille de l'entrée).
- Notons par $T(n)$ le coût de l'algorithme en nombre de comparaisons.
- Nous avons : $T(1) = 2$ et $T(n) \leq T(\lceil n/2 \rceil) + 2$, pour tout $n > 1$.
- D'après le théorème général, $T(n) = O(\log_2 n)$. Il s'agit donc d'un algorithme logarithmique.

# 3- Algorithmes de tri

## A) Algorithme de tri par sélection du minimum :

**Principe** :

- À chaque itération, on recherche le plus petit élément dans la partie non encore triée du tableau.
- On échange cet élément avec le premier élément dans la même partie du tableau.

**Algorithme** :

**Fonction** TriSélection(*A* : **Tableau d'Entier**, *n* : **Entier**)

**Variables** *i*, *j*, *k* : **Entier**; *// j indice du minimum*

**Début**

**Pour** *i* **:=** 1 **à** *n* - 1 **Faire**

*j* **:=** *i*;

**Pour** *k* **:=** *i* + 1 **à** *n* **Faire**

**Si** (*A*[*k*] < *A*[*j*]) **Alors** *j* **:=** *k*; **FinSi**

**FinPour** {*k*}

**Si** (*j* ≠ *i*) **Alors Echanger**(*A*[*i*], *A*[*j*]) **FinSi**

**FinPour** {*i*}

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

Calculons la complexité (exacte) :

- Notons par $T(n)$ le coût de l'algorithme en nombre de comparaisons ($n$ taille du tableau).
- $$T(n) = \sum_{i=1}^{n-1} \sum_{k=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = (n-1) + (n-2) + \ldots + 1 = \frac{(n-1)n}{2} = \Theta(n^2).$$

## B) Algorithme de tri par insertion :

**Principe** :

- À chaque itération, on prend le premier élément dans la partie non encore triée du tableau (la clé).
- On cherche la place de la clé dans la partie déjà triée du tableau, en commençant par la droite de cette partie.
- Une fois cette place trouvée, on y insère la clé et en décale vers la droite les autres éléments de la partie triée dont l'indice est plus grand ou égal à la position.

**Algorithme** :

**Fonction** TriInsertion(*A* : **Tableau d'Entier**, *n* : **Entier**)

**Variables** *i*, *j*, *k* : **Entier**; *// j indice du minimum*

**Début**

**Pour** *i* **:** = 2 à *n* **Faire**

*Clé* **:** = *A*[*i*];

*j* **:** = *i* - 1;

**Tant Que** ((*j* >= 1) **ET** (*Clé* < *A*[*j*])) **Faire**

*A*[*j* + 1] **:** = *A*[*j*]; *// Décalage*

*j* **:** = *j* - 1;

**FinTQ**

*A*[*j* + 1] **:** = *Clé*; *// Insertion de la clé*

**FinPour**

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

- Dans le meilleur des cas :
  - Configuration qui donne le meilleur des cas : le tableau *A* est trié dans l'ordre croissant.
  - $Min_{TriInsertion}(n) = \sum_{i=2}^{n} 1 = n - 1 = \Omega(n).$
- Dans le pire des cas :
  - Configuration qui donne le pire des cas : le tableau *A* est trié dans l'ordre décroissant.
  - $Max_{TriInsertion}(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n} (i-1) = 1 + 2 + ... + (n-1) = \frac{(n-1)n}{2} = O(n^2).$

## C) Algorithme de tri à bulles :

**Principe** :

- À chaque itération, on compare dans la partie non encore triée du tableau, un élément et son suivant, en commençant par le premier élément dans cette partie.
- Si les deux éléments successifs sont dans le bon ordre, on progresse.
- Sinon, on les permute.
- De cette façon on ramène le plus grand élément à la dernière position dans la partie non encore triée du tableau.

**Algorithme** :

**Fonction** TriBulles(*A* : **Tableau d'Entier**, *n* : **Entier**)

**Variables** *i*, *j*, *k* : **Entier**; *// j indice du minimum*

**Début**

**Pour** *i* **:** = 1 **à** *n* - 1 **Faire**

**Pour** *j* **:** = 1 **à** (*n* - *i*) **Faire**

**Si** (*A*[*j*] > *A*[*j* + 1]) **Alors Echanger**(*A*[*j*], *A*[*j* + 1]); **FinSi**

**FinPour**

**FinPour**

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

Calculons la complexité (exacte) :

- Notons par $T(n)$ le coût de l'algorithme en nombre de comparaisons ($n$ taille du tableau).
- $T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n-1} (n-i) = \frac{(n-1)n}{2} = \Theta(n^2).$
  - $Max_{TriInsertion}(n) = \sum_{i=2}^{n} \sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n} (i-1) = 1 + 2 + ... + (n-1) = \frac{(n-1)n}{2} = \mathrm{O}(n^2).$

## D) Algorithme de tri par fusion :

**Principe** :

Il s'agit d'un algorithme suivant une stratégie de type diviser pour régner. L'algorithme procède comme suit :

- On divise le tableau en deux parties égales.
- On effectue deux appels récursifs pour trier les deux parties du tableau.
- On fusionne les deux parties triées du tableau pour former un seul tableau trié (procédure de fusion).

**Algorithmes** :

**Fonction** TriFusion(*A* : **Tableau d'Entier**, *deb*, *fin* : **Entier**)

*// L'algorithme de tri par fusion*

**Variables** *milieu* : **Entier**;

**Début**

**Si** (*deb* < *fin*) **Alors**

$$milieu := \left\lfloor \frac{deb + fin}{2} \right\rfloor;$$

**TriFusion**(*A*, *deb*, *milieu*);

**TriFusion**(*A*, *milieu* + 1, *fin*);

**Fusion**(*A*, *deb*, *milieu*, *fin*);

**FinSi**

**Fin**

**Fonction** Fusion(*A* : **Tableau d'Entier**, *deb*, *milieu*, *fin* : **Entier**)

*// Procédure de fusion*

**Variables** *i*, *j*, *k* : **Entier**;

*B* : **Tableau**;

**Début**

**Pour** *k* : = *deb* **à** *milieu* **Faire**

*B*[*k*] : = *A*[*k*];

**FinPour**

**Pour** *k* : = *milieu* + *1* **à** *fin* **Faire**

*B*[*k*] : = *A*[*fin* - *k* + *milieu* + 1];

**FinPour**

*i* : = *deb*; *j* : = *fin*; *k* : = 1;

**Tant Que** (*i* < *j*) **Faire**

**Si** (*B*[*i*] < *B*[*j*]) **Alors**

*A*[*k*] : = *B*[*i*]; *i* : = *i* + 1;

**Sinon**

*A*[*k*] : = *B*[*j*]; *j* : = *j* - 1;

**FinSi**

*k* : = *k* + 1;

**FinTQ**

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

Calculons la complexité de cet algorithme dans le pire des cas :

- Soit $n = fin - deb + 1$ la taille du tableau (c'est la taille de l'entrée) et notons par $T(n)$ le coût de l'algorithme en nombre de comparaisons.
- $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + (n-1)$, pour $n$ assez grand. En fait, la procédure de fusionne nécessite (n - 1) comparaisons.

- D'après le théorème général, $T(n) = \mathrm{O}(n \log_2 n)$.

## E) Algorithme de tri rapide :

**Principe** :

Il s'agit également d'un algorithme suivant une stratégie de type diviser pour régner. L'algorithme procède comme suit :

- On partitionne le tableau de façon à avoir deux parties vérifiant la propriété suivante : tous les éléments de la première partie sont inférieurs à un élément *p* (appelé le pivot), et tous les éléments de la deuxième partie sont supérieurs à ce même élément *p*. Ainsi, l'élément *p* se trouvera dans sa place convenable.
- On effectue deux appels récursifs pour trier de la même façon les deux parties du tableau.

**Algorithmes** :

```
Fonction TriRapide(A : Tableau d'Entier, deb, fin : Entier)
// L'algorithme de tri rapide
Début
  Si (deb < fin) Alors
                    pivot := Partition(A, deb, fin);
                    TriRapide(A, deb, pivot - 1);
                    TriRapide(A, pivot + 1, fin);
  FinSi
Fin

Fonction Partition(A : Tableau d'Entier, deb, fin : Entier)
// Procédure de partitionnement
Variables     i, pivot, x : Entier;
              B : Tableau;
Début
  x := A[fin];
  pivot := deb - 1;
  Pour i := deb à fin - 1 Faire
      Si (A[i] ≤ x) Alors
                    pivot := pivot + 1;
                    Echanger(A[pivot], A[i]);
      FinSi
  FinPour
```

**Echanger**(*A*[*pivot* + 1], *A*[*fin*]);

**Retourner**(*pivot* + 1);

**Fin**

**Exercice** : Donner la preuve de cet algorithme, puis le programmer en C++.

**Complexité en nombre de comparaisons** :

Calculons la complexité de cet algorithme dans le pire des cas :

- Soit $n = fin - deb + 1$ la taille du tableau (c'est la taille de l'entrée) et notons par $T(n)$ le coût de l'algorithme en nombre de comparaisons.
- $T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$, pour $n$ assez grand. En fait, la procédure de partitionnement nécessite un nombre de comparaisons en $\Theta(n)$.
- Montrons, par substitution que $T(n) = \mathrm{O}(n^2)$ :

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n)$$

$$T(n) \le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n)$$

$$T(n) \le c \times \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n)$$

$$T(n) \le c \times (n-1)^2 + \Theta(n)$$

$$T(n) \le cn^2 + c'n \le 2\max(c, c')n^2 = Kn^2, \text{ où } K = \max(c, c') > 0.$$

- En conclusion, l'algorithme de tri rapide est en $\mathrm{O}(n^2)$.

Calculons la complexité de l'algorithme dans le meilleur des cas :

- Dans le cas où le partitionnement produit des parties de taille inférieure ou égale à ($n$ / 2), le temps d'exécution de l'algorithme serait plus rapide. En effet : $T(n) \le 2T(n/2) + \Theta(n)$, pour $n$ assez grand. D'après le théorème général, une telle fonction serait en $\mathrm{O}(n \log_2 n)$.

**Remarque** : le temps d'exécution moyen du tri rapide est plus proche du cas optimal (meilleur des cas) que du cas le plus défavorable (pire des cas).

## Mots clés du chapitre :

Algorithme de recherche séquentielle

Algorithme de recherche binaire (dichotomique)

Algorithme de tri par sélection

Algorithme de tri par insertion

Algorithme de tri à bulles

Algorithme de tri par fusion

Algorithme de tri rapide