CHAPITRE 1

RECHERCHE

Comment trouver
des données

RECHERCHES

- Introduction
- Recherche de base
- Recherche dichotomique

INTRODUCTION

- Les traitements informatiques nécessitent constamment de classer les informations qu'elles traitent dans un certain ordre.
- Or il existe de nombreuses méthodes de recherche et de tri, selon plusieurs paramètres :
  - ✓ Le volume d'informations à trier. Par exemple, pour des petits volumes, une méthode simple ira plus vite qu'un tri rapide mais complexe. De même, pour de gros volumes, on pourra changer de méthode en cours de tri pour optimiser le temps.
  - ✓ L'ordre dans lequel sont rangées les données au départ. En particulier, les tris les plus rapides supposent un certain ordre de départ, sous peine de devenir très lents. C'est pourquoi des heuristiques (méthode pour déterminer le meilleur résultat sans une analyse complète, comme les calculs d'itinéraires) sont parfois utilisées pour déterminer la meilleure méthode en fonction de la répartition des données.
  - ✓ Le type des données influe sur la méthode de comparaison entres elles et donc sur la méthode de tri. Ainsi, ***trier des nombres est plus rapide*** que trier des mots, des images ou autres, car ils ne nécessitent qu'une seule comparaison pour leur donner un ordre. C'est pourquoi on cherchera toujours à les utiliser de préférence.
- **Note :** on prendra dans les exemples suivants un tableau défini comme suit :

```
T : tableau [ 1 ; N ] d'entiers ;
```

LA RECHERCHE DE BASE (1/2)

- Rechercher des données dans un tableau est très simple à priori, il suffit de parcourir le tableau jusqu'à ce qu'on trouve l'élément recherché. Mais on l'utilise tellement souvent qu'il est important des trouver des méthodes pour accélérer le processus. Voilà les 2 méthodes de base :
- Algorithme de recherche de la 1ère occurrence de l'élément recherché :

```
Fonction RecherchePremier( T : Tableau[1..N] d'Entiers ;
                           élément : Entier) : Entier
   Variable
      i : Entier = 1
   Début
   Tant Que (i <= N ) ET (élément ≠ T[i])  Faire
      i ← i + 1
   Fin Tant Que
   Si  i<= N Alors Retourner i    // on renvoie la position
             Sinon Retourner -1   // non trouvé
   Fin Si
Fin
```

LA RECHERCHE DE BASE (2/2)

- Algorithme de recherche de toutes les occurrences de l'élément recherché :
- Soit l'algorithme :

```
Procédure RechercheTout( T : Tableau[1..N] d'Entiers ;
                         élément : Entier)
    Variable
        i : Entier
    Début
    Pour i de 1 à N Faire
        Si élément = T[i] Alors Afficher i // on affiche la position
                           // PS : on pourrait les retenir dans un tableau
        Fin Si
    Fin Pour
Fin
```

- Ces deux méthodes sont simples, mais le temps de recherche est en **O(N)** comparaisons pour N enregistrements. En ayant trié le tableau auparavant, on peut aller plus vite :

LA RECHERCHE DICHOTOMIQUE

- A partir du moment où un tableau est trié, on peut tester la donnée du milieu. Si ce n'est pas la bonne, on restreint alors la recherche, selon le cas, à la moitié inférieure ou supérieure du tableau. Et on recommence sur cette partie jusqu'à trouver.
- Le temps de recherche maximum est alors en $O(Log_2 N)$ comparaisons, pour N enregistrements.
- Soit l'algorithme :

```
Fonction Dichotomie(T:Tableau[1..N] d'Entiers ; élément:Entier) : Entier
Variable
    deb : Entier = 1,
    fin : Entier = N,
    mil : Entier = (deb + fin) / 2
Début
    Tant Que (deb <= fin) ET (élément ≠ T[mil])  Faire
        Si élément < T[mil] Alors fin ← mil - 1
                            Sinon deb ← mil + 1
        Fin Si
        mil ← (deb + fin) / 2
    Fin Tant Que
    Si  deb <= fin  Alors  Retourner mil
                   Sinon  Retourner  -1   // non trouvé
    Fin Si
Fin
```

CHAPITRE 2

TRIS

Comment trier des données

TRIS

- Introduction
- Tri par insertion
- Tri par sélection
- Tri à bulle
- Tri rapide : QuickSort

INTRODUCTION

- Les deux principaux facteurs déterminant l'efficacité d'une méthode de tri sont le temps d'exécution et l'espace mémoire utilisé.
- Le tri le plus rapide consiste à réserver une case par valeur possible, puis de noter chaque valeur qui apparaît. Mais cette méthode est irréalisable en pratique car l'espace requis est trop important.
- En général, l'espace requis étant rarement important, on privilégie donc le temps d'exécution et, en particulier, le nombre de comparaisons car c'est l'opération qui prend théoriquement le plus de temps.
- Cependant, dans le cas de données de taille importante (images, sons, grosses bases de données, ...) ou d'un accès lent à ces données, on prêtera attention au nombre de permutations entre ces données.
- **Note :** on prendra dans les exemples suivants un tableau défini comme suit que l'on triera en ordre croissant :

```
T : tableau [ 1 ; N ] d'entiers ;
```

LE TRI PAR INSERTION

- Le tri par insertion est l'approche intuitive par excellence.
- **Principe :**
  - ✓ On considère les (i-1) premiers éléments triés et on sauvegarde l'élément i.
  - ✓ On recherche la place de l'élément i dans les (i-1) précédents en décalant progressivement à droite les valeurs.
  - ✓ On remet l'élément i à sa nouvelle place.
  - ✓ On réitère sur les éléments suivants.
- **Exemple :**

2	5	3	1	4

2	3	5	1	4

- **Étape 1 :** on cherche la place de 3 dans (2,5). On le remonte donc en tête de tableau en décalant 5. Enfin on place 2 à sa place 2.

1	2	3	5	4

- **Étape 2 :** on recommence pour 1, puis 4.

1	2	3	4	5

LE TRI PAR INSERTION : ALGORITHME

```
Pour i allant de 2 à N Faire
     j ← i - 1
     x ← T[i]
     Tant Que  ( j>0  ET  T[j]>x )  Faire
          T[j+1] ← T[j]              // on cherche la place de x
          j ← j-1                    // en décalant les éléments
     Fin Tant Que
     T[j+1] ← x              // on recopie T[i] à sa place
Fin Pour
```

- Complexité :
  - ✓ Quel que soit l'ordre du tableau initial, le nombre de tests et d'échanges reste le même et on doit aller jusqu'au bout.
  - ✓ On effectue 1 tests pour placer le 1er élément du tableau trié, 2 tests pour le 2ème, et ainsi de suite. Soit : 1+2+...+(N-1)+(N-2) = N(N-1)/2. On effectue en plus autant d'échanges.
  - ✓ La complexité est identique dans tous les cas, soit de l'ordre de

    $O(N^2)$ comparaisons et **$O(N^2)$ échanges**
  - ✓ Par exemple, pour un ordinateur qui effectue 10^6 tests par seconde on a :

N	10^3	10^6	10^9
temps	1s	11,5 jours	32 000 ans

LE TRI PAR SÉLECTION

- Il permet de minimiser le nombre d'échanges.
- **Principe :**
  - ✓ Ayant trié les (i-1) premiers éléments du tableau.
  - ✓ A l'étape i, on sélectionne le plus petit élément parmi les (n - i +1) éléments du tableau les plus à droite.
  - ✓ Ensuite, on l'échange si nécessaire avec l'élément i du tableau.
  - ✓ On recommence pour l'élément (i+1), etc.
- **Exemple :**

Étape 1 : on cherche le plus petit parmi les 5 éléments du tableau. On l'identifie en 3ème position, et on l'échange alors avec l'élément 1 :

9	4	1	7	3

1	4	9	7	3

- **Étape 2 :** on cherche le plus petit élément, mais cette fois à partir du deuxième élément. On le trouve en dernière position, on l'échange avec le deuxième :

1	3	9	7	4

1	3	4	7	9

LE TRI PAR SÉLECTION : ALGORITHME

```
Pour i variant de 1 à N-1 Faire
      min ← i                    // min contient l'indice du minimum
      Pour j variant de i+1 à N Faire // on cherche le minimum
            Si  T[j] < T[min]  Alors  min ← j
            Fin Si
      Fin Pour
      Si  i ≠ min  Alors     // on échange T[min] et T[i]
            tmp ← T[i]
            T[i] ← T[min]
            T[min] ← tmp
      Fin Si
Fin Pour
```

- **Complexité :**
  - ✓ Ici aussi, le nombre de tests reste le même, mais on ne fait qu'un échange par étape et on doit aller jusqu'au bout.
  - ✓ On effectue N-1 tests pour trouver le 1er élément du tableau trié, N-2 tests pour le 2ème, et ainsi de suite. Soit : (N-1)+(N-2)+...+1 = N(N-1)/2. On effectue en plus (N-1) échanges.
  - ✓ La complexité est identique dans tous les cas, soit de l'ordre de

$O(N^2)$ comparaisons et **$O(N)$ échanges**

LE TRI À BULLE

- C'est le plus simple à implémenter et surtout il peut s'arrêter avant la fin normale.
- **Principe :**
  - ✓ A l'étape i, les i derniers éléments du tableau sont triés.
  - ✓ On prend alors les (n-i) premiers éléments du tableau.
  - ✓ On compare les éléments adjacents du tableau et on les échange pour les mettre en ordre 2 à 2. On remonte ainsi la plus grande valeur de la partie de tableau considérée
  - ✓ On recommence l'opération jusqu'à ce qu'il n'y ait plus d'échange sur une ligne.
- **Exemple d'une phase où 9 est en place :**

On compare 7,4 puis 7,1 puis 7,3.
On échange à chaque fois les valeurs.
On remonte ainsi 7 en avant dernière position :

7	4	1	3	9

4	7	1	3	9

4	1	7	3	9

4	1	3	7	9

LE TRI À BULLE : ALGORITHME

```
i ← N - 1
Répéter
      echange ← faux
      Pour j variant de 1 à i Faire
            Si T[j]>T[j+1] Alors     // Échange de T[j] et T[j+1]
                  tmp ← T[j]
                  T[j] ← T[j+1]
                  T[j+1] ← tmp
                  echange ← vrai
            Fin Si
      Fin Pour
      i ← i - 1
Jusqu'à i ≤ 0 OU echange = faux
```

- **Complexité :**
  - ✓ On effectue N-1 tests et échange pour amener le plus grand élément à la fin du tableau, puis N-2 au 2ème tour pour le second, et ainsi de suite. La complexité est donc de l'ordre de

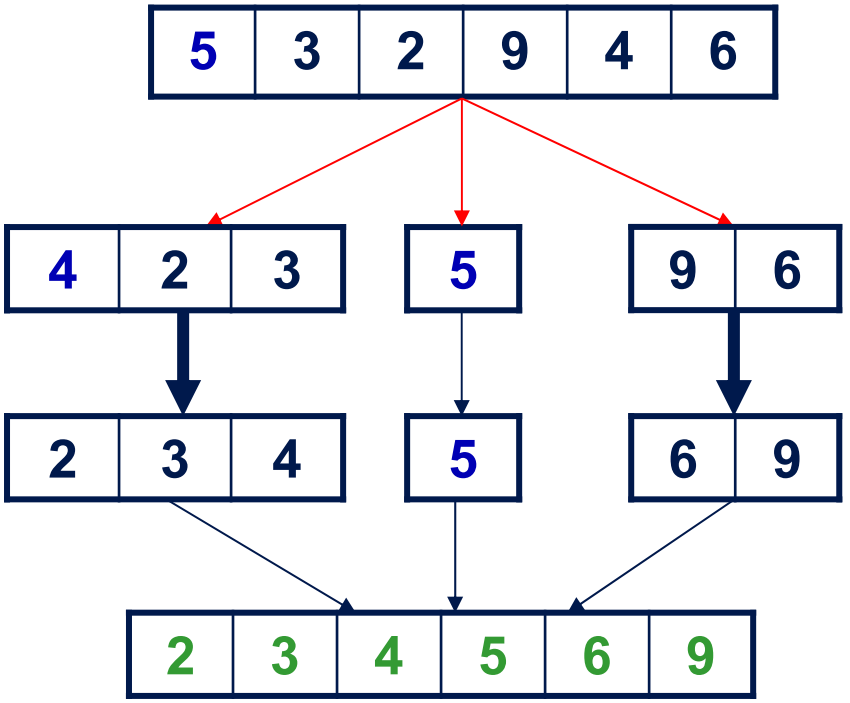
    $O(N^2)$ comparaisons et **$O(N^2)$ échanges**
  - ✓ Mais le tri peut s'arrêter avant la fin s'il n'y a aucun échange pendant la boucle intérieure. Mais cela reste aléatoire.

LE TRI RAPIDE

- Le **tri rapide**, ou **Quicksort**, est un tri récursif basé sur l'approche "diviser pour régner", ce qui consiste à décomposer un problème d'une taille donnée à des sous problèmes similaires mais de taille inférieure faciles à résoudre.

- **Description du tri rapide :**

  - ✓ On considère un élément du tableau qu'on appelle pivot
  - ✓ On partitionne le tableau en 2 sous tableaux : les éléments inférieurs ou égaux à pivot et les éléments supérieurs à pivot. on peut placer ainsi la valeur du pivot à sa place définitive entre les deux sous tableaux.
  - ✓ On répète récursivement ce partitionnement sur chacun des sous tableaux crées jusqu'à ce qu'ils soient réduits à un à un seul élément.



LE TRI RAPIDE : ALGORITHME (1/2)

```
Procédure TriRapide(T : Tableau d'Entiers par adresse, deb, fin : Entier)
Variable pivot : Entier  // indice du pivot
Début
     Si deb < fin  Alors
          pivot ← Partition(T, deb, fin)
          TriRapide(T, deb, pivot - 1)
          TriRapide(T, pivot + 1, fin)
     Fin Si
Fin Procédure
```

- A chaque étape de récursivité on partitionne un tableau *T[deb..fin]* en deux sous tableaux *T[deb..pivot-1]* et *T[pivot+1..fin]* tel que chaque élément de *T[deb..pivot-1]* soit inférieur ou égal à chaque élément de *T[pivot+1..fin]*.
- L'indice final du *pivot* est calculé pendant la fonction de partitionnement. On fixe arbitrairement ici son indice de départ à l'indice de début de la partition de tableau à trier. Certains algorithmes utilisent des heuristiques pour améliorer cette position selon sa valeur puisqu'elles va fortement influencer une bonne partition des données et donc la rapidité du tri..
- Puis on relance la récursivité sur chaque sous partie de tableau.

LE TRI RAPIDE : ALGORITHME (2/2)

```
Fonction Partition(T : Tableau d'Entier par adresse; deb, fin) : Entier
Variables inf, sup : Entier
          valPvot : Entier  // valeur du pivot
Début
    valPivot ← T[deb]  // on fixe le pivot à la case d'indice deb
    inf ← deb + 1    // indice de parcours à partir du début du tableau
    sup ← fin          // indice de parcours à partir de la fin du tableau
    Tant Que (inf ≤ sup) Faire
        Tant Que (inf ≤ fin et T[inf] ≤ valPivot ) Faire inf ← inf + 1 Fin Tant Que
        Tant Que (sup ≥ deb et T[sup] > valPivot ) Faire sup ← sup - 1 Fin Tant Que
        Si (inf < sup) Alors
            Échanger(T[inf], T[sup])
            inf ← inf + 1
            sup ← sup - 1
        Fin Si
    Fin Tant Que
    Échanger(T[sup], T[deb]) // procédure d'échange des cases du tableau
    Retourner sup              // on retourne l'indice trouvé pour le pivot
Fin Fonction
```

LE TRI RAPIDE : COMPLEXITÉ

- La complexité du tri rapide en moyenne est en **$O(N \log_2 N)$ comparaisons et échanges** ce qui est d'autant meilleur que **$O(N^2)$** quand N grandit (plus de 20 valeurs, mais on considère que la complexité de mise en œuvre porte cette valeur à 100).
- Le pire des cas se produit quand le pivot est à chaque choix le plus petit élément du tableau (tableau déjà trié). Dès lors, cette complexité revient à **$O(N^2)$**. De plus, la complexité de sa mise en place en font un tri plus lent que ceux vus précédemment.
- C'est pourquoi le choix du pivot influence largement les performances du tri rapide. Pour cela, on utilise des heuristiques pour le choisir. Ce sont des algorithmes qui permettent de déterminer le meilleur choix sans connaître le résultat exact et en beaucoup moins de temps, comme par exemple les algorithmes de calcul d'itinéraire.
- En conséquence, différentes versions du tri rapide sont proposées dans la littérature pour rendre le pire des cas le plus improbable possible, ce qui rend cette méthode la plus rapide en moyenne parmi toutes celles utilisées.
- Le tri par tas, Heapsort, basé sur des arbres binaires, non vu ici, est aussi en complexité **$O(N \log_2 N)$** mais il est constant en temps. Cependant, il reste plus lent en moyenne.

CHAPITRE 3

LA PROGRAMMATION DE REQUÊTES SQL

Comment sont programmées les requêtes SQL de bases de données

INTRODUCTION (1/2)

- L'exécution des requêtes SQL de sélection est basée sur l'édition du contenu d'un tableau de structure (le principe reste le même pour d'autres structures).
- Une boucle parcourt donc le tableau et, pour chaque enregistrement, on affiche les champs demandés dans le SELECT. S'il y a des conditions, on les valide avant pour afficher.
- La difficulté intervient lorsqu'il faut afficher des résultats sur plusieurs enregistrements, c'est-à-dire lorsque l'on fait des opération de totalisation avec éventuellement des ruptures en fonction d'un champ.
- On redéfinit ces 2 termes :
- **Totalisation :** récapitulatif des données avec calculs éventuels sur l'ensemble des données. L'algorithme est similaire à l'édition en rajoutant les calculs.
- **Rupture :** méthode permettant de faire des sous-totaux ou des affichages regroupés selon les valeurs de certains champs.

INTRODUCTION (2/2)

- L'exécution des requêtes SQL est donc basé sur ces 2 principes de totalisation et de rupture. Cela donne 3 algorithmes possibles en fonction de la priorité que l'on donne aux 2 options suivantes :
  - ✓ Rapidité : on ne parcourt qu'une fois les données et on sauvegarde les informations récapitulatives dans des structures temporaires. Il n'y a ainsi qu'un seul parcours de la base de données, mais les informations retenues sont proportionnelles à la taille de la base. On peut minimiser la taille de ces informations en travaillant sur un tableau trié, mais s'il ne l'est pas, c'est une grosse perte de temps.
  - ✓ Minimum d'espace : on parcourt une fois l'ensemble des données pour chaque information à récupérer. Celles-ci n'ont donc pas besoin d'être retenues, mais le temps d'exécution peut être beaucoup plus long.
- **Exemple de base :** on retient dans une structure les chiffres d'affaires de commerciaux
-

```
Commercial : Structure
                nom : String
                region : Entier   // 1=Nord ; 2=Sud ; 3=Ouest ; 4=Est
                ca : Entier       // chiffre d'affaires
Fin Structure
Tcom : Tableau [ 1 ; MAX ] de Commercial
```

ALGORITHME 1

- Dans cette 1ère méthode, on trie le tableau par rapport au champ de rupture (ici la région)
- On parcourt le tableau et on affiche le résultat à chaque changement du champ de rupture
- Avantage : 1 seul parcours, pas d'espace mémoire nécessaire
- Inconvénient : nécessite un tri auparavant

```
Procédure Méthode1
VAR
      somme, reg : Entier
      i : Entier
DEBUT
      Tri(Tcom, region) // On trie le tableau avant par rapport à la région
      somme ← Tcom[1].ca    // On initialise la somme et la région
      reg ← Tcom[1].region  // à la valeur de la 1ère case
      Pour i allant de 2 à MAX Faire
          Si Tcom[i].region = reg  Alors
               somme ← somme  +  Tcom[i].ca
          Sinon     // Changement de région → on affiche la précédente
               Aff ("CA de la Région", i, " : ", somme)
               somme ← Tcom[i].ca // On réinitialise la somme et la région
               reg ← Tcom[i].region
          Fin Si
      Fin Pour
      Afficher ("CA de la Région", i, " : ", somme)
FIN
```

ALGORITHME 2

- Dans cette 2ème méthode, on fait 1 parcours du tableau par région
- Avantage : cette ne nécessite pas d'espace mémoire supplémentaire
- Inconvénient : on doit faire 1 par parcours par région, donc très long, d'autant plus que le nombre peut être important

```
Procédure Méthode2
VAR
      somme, reg : Entier
      i : Entier
DEBUT
      Pour reg allant de 1 à NBREGION Faire
            somme ← 0
            Pour i allant de 1 à MAX Faire
                  Si Tcom[i].region = reg  Alors
                        somme ← somme  +  Tcom[i].ca
                  Fin Si
            Fin Pour
            Aff ("CA de la Région", reg, " : ", somme)
      Fin Pour
FIN
```

ALGORITHME 3

- Dans cette 3ème méthode, on fait 1 seul parcours du tableau et on met à jour un tableau des sommes pour chaque région
- Avantage : 1 seul parcours du tableau
- Inconvénient : 1 tableau de taille la cardinalité du champ de rupture (ici la région)

```
Procédure Méthode3
VAR
      somme : Tableau [1 ; NBREGION ] d'Entier
      i, reg : Entier
DEBUT
      Pour i allant de 1 à NBREGION Faire
            somme[i] ← 0
      Fin Pour
      Pour i allant de 1 à MAX Faire
            reg ← Tcom[i].reg
            somme[ reg ] ← somme[ reg ] + Tcom[i].ca
      Fin Pour
      Pour i allant de 1 à NBREGION Faire
            Aff ("CA de la Région", i, " : ", somme[reg])
      Fin Pour
FIN
```