CH.3 ALGORITHMES DE TRI

- 3.1 Les tris quadratiques
- 3.2 Les tris en n log n
- 3.3 Le tri rapide
- 3.4 Les bornes inférieures

3.1 Les tris quadratiques

On dispose de n données en mémoire centrale (table, liste chaînée) qu'il s'agit d'ordonner de manière croissante.

On peut supposer que ces données sont des nombres entiers.

Trois classes d'algorithmes :

- en temps $O(n^2)$ dans le cas le pire et en moyenne ;
- en temps $O(n \log n)$ dans le cas le pire et en moyenne ;
- en temps $O(n^2)$ dans le cas le pire et en $O(n \log n)$ en moyenne.

Une complexité en $n \log n$ est le minimum possible, aussi bien dans le cas le pire qu'en moyenne.

Les tris les plus intuitifs sont quadratiques en temps.

Tri par **sélection** :
On cherche le minimum dans la liste.
On échange ce minimum avec le premier élément de la liste.
On recommence avec la liste restante, jusqu'à épuisement.

On peut, alternativement, prendre le plus grand et l'échanger avec le dernier.

```
fonction TRI_SELECTION(tableau t[n])
{
   pour i=1 à n-1 faire
   {
      min=i;
      pour j=i+1 a n faire si t[j] < t[min] alors min=j;
      echanger(t[i], t[min]);
   }
}
```


Complexité de l'algorithme.
En temps :
la boucle intérieure demande un temps $A(n - i) + B$;
cette boucle est parcourue pour i = 1, 2, ... , $n - 1$.
Le temps total est donc $An(n-1)/2 + Bn = O(n^2)$.
Toutes les opérations sont à effectuer quel que soit le tableau.
Cette complexité est donc la même pour tous, et donc aussi en moyenne.
La place requise (version itérative, tri sur place) en plus du tableau est constante.

Algorithme intuitif, facile à écrire.

Tri par **insertion** :
On suppose que les $i - 1$ premiers éléments de la liste sont déjà classés.
On insère à sa place le i -ième élément parmi les $i - 1$ premiers ; de la sorte, les i premiers éléments sont classés.
On itère jusqu'à insérer le n-ième.

```
Fonction TRI_INSERTION(tableau t[n])
{
   pour i=2 a n faire
   {
      k=i-1; temp=t[i];
      tant que (k >=1) et (temp < t[k]) faire
      {
         t[k+1]=t[k]; k--;
      }
      t[k+1]=temp;
   }
}
```


Complexité de l'algorithme.
En temps :
dans le cas le pire, on fait, dans la boucle intérieure, $i - 1$ comparaisons et autant de décalages : temps $A(i - 1) + B$;
cette boucle est parcourue pour $i = 2, \dots, n$.
Le temps total dans le cas le pire est donc $O(n^2)$.
En moyenne, le nombre de comparaisons est intuitivement divisé par 2. Cela ne change pas l'ordre de grandeur.

La place requise (version itérative, tri sur place) en plus du tableau est constante comme précédemment.

Algorithme intuitif comme le précédent. En moyenne, moins de comparaisons, mais plus de décalages ici que d'échanges auparavant.

Tri par **bulles** :

On balaye la liste en échangeant deux éléments consécutifs s'ils sont dans le mauvais ordre. Le plus grand élément de la liste est donc repoussé à la fin.

On recommence à partir du début, avec les $n-1$ premiers éléments et ainsi de suite.

Cela ressemble un peu au tri par sélection, à ceci près qu'on utilise la recherche du maximum pour corriger quelques autres inversions.

```
Fonction TRI_BULLES(tableau t[n])
{
   fini=faux;
   pour i=n a 2 faire si fini == faux alors
   {
      fini=vrai;
      pour j=1 a i-1 faire
      si t[j] > t[j+1] alors
      {
         echanger(t[j], t[j+1]);
         fini=faux;
      }
   }
}
```


Le nom vient de ce que les grands éléments sautent par dessus les plus petits jusqu'à la fin du tableau, comme des bulles...

Complexité de l'algorithme.

En temps :

Le raisonnement est le même que pour le tri par insertion.

Le temps total dans le cas le pire est donc $O(n^2)$.

Si on trie une permutation, le nombre d'échanges faits est le nombre d'inversions. Le nombre de comparaisons est au moins ce nombre. Or le nombre moyen d'inversions est en $O(n^2)$. Donc le nombre moyen de comparaisons est au moins quadratique, ainsi que le temps moyen.

La place requise (version itérative, tri sur place) en plus du tableau est constante comme précédemment.

Algorithme “local”.

3.2 Les tris en *n* log *n*

Les tris présentés sont des tris dont le temps dans les cas le pire et en moyenne sont en $O(n \log n)$.

Le premier est une illustration du principe "**diviser pour régner**".
On ramène la résolution d'un problème de taille *n* à la résolution du même problème sur deux occurrences de taille divisée par 2. Ceci ne réduit le temps que si la complexité du rassemblement des solutions des problèmes de taille plus petite est simple.

Parmi les exemples les plus significatifs, on trouve, la recherche dichotomique, l'exponentiation rapide, le tris par tas et la transformée de Fourier rapide.

Tri par **fusion** :
On divise la liste en deux parts égales. On trie chacune de ces deux listes. Finalement, on fusionne ces deux listes triées.
La première fonction réalise la fusion entre deux listes $t1$ et $t2$ et place le résultat dans la liste *t*. On suppose $n1 + n2 = n$

```
Fonction FUSION(tableau t[n], tableau t1[n1], tableau t2[n2])
{
   j=1; k=1;
   pour i=1 a n faire
   {
      si (j <= n1) et ((k > n2) ou (t1[j] <= t2[k])) alors
      {
         t[i]=t1[j]; j++;
      }
      sinon
      {
         t[i]=t2[k]; k++;
      }
   }
}
```

Le temps pour effectuer la fusion est proportionnel à n : chaque itération fait écrire une nouvelle valeur dans la table, sauf quand elle s'y trouve déjà (lorsque la première sous-liste est vide).

L'algorithme de tri est maintenant très simple à écrire récursivement.

```
Fonction TRI_FUSION(tableau t[n])
{
   si n == 1 alors retour;
   n1=n/2; n2=n-n1;
   pour i=1 a n1 faire t1[i]=t[i];
   pour i=1 a n2 faire t2[i]=t[n1+i];
   TRI_FUSION(t1[n1]); TRI_FUSION(t2[n2]);
   FUSION(t[n], t1[n1], t2[n2]);
}
```


Pour calculer le temps nécessaire, on suppose que $n = 2^k$. (On peut imaginer qu'on complète la liste par des éléments très grands.)

La fusion requiert un temps linéaire. Il en est de même pour la copie dans les tableaux temporaires.

On appelle récursivement le tri sur des listes de taille moitié.

Donc $T(2^k) = 2\ T(2^{k-1}) + C\ 2^k$. En posant $U(k) = T(2^k)$, on trouve $U(k) = 2\ U(k-1) + C\ 2^k$. La solution est $U(k) = C\ k\ 2^k + A\ 2^k$.

En revenant à n, on obtient finalement $T(n) = O(n \log n)$.

En ce qui concerne la place, on peut remarquer que les deux appels récursifs sont consécutifs. La même place va donc être utilisée pour ces deux appels. Donc $P(2^k) = P(2^{k-1}) + 2^k$, le terme constant venant de l'allocation de $t1$ et $t2$.

On obtient à la fin $P(n) = O(n)$. On peut diminuer la place en faisant la fusion sur place (nécessite des décalages dans la table.)

Le tri par **tas** :

On a déjà vu l'algorithme dans un chapitre précédent. Le temps nécessaire est ici encore en $O(n \log n)$ dans le cas le pire et la place constante.
L'algorithme transforme sur place une liste en tas, puis extrait successivement le premier élément, qui est le maximum, et réorganise le tas.

Dans le cas de ces deux algorithmes, nous verrons pourquoi le temps moyen est ici encore en $O(n \log n)$.

La troisième catégorie est celle des algorithmes dont le temps est quadratique dans le pire des cas, mais en $n \log n$ en moyenne.

3.3 Le tri rapide

Le tri par **arbre binaire de recherche** :
Étant donnée une liste de valeurs à trier, on peut commencer à fabriquer un ABR contenant ces données.
Puis, on extrait le minimum (noeud obtenu en descendant le plus à gauche).
On rétablit la condition d'ABR et on recommence.
On a vu que, si la liste est suffisamment aléatoire, la hauteur de l'ABR est en $O(\log n)$ et sa construction prend un temps $O(n \log n)$.
Ceci est vrai en fait en moyenne. Dans ce cas, toutes les opérations suivantes sont en $O(\log n)$ et répétées n fois.
D'où le temps $O(n \log n)$ en moyenne.
Pour la place, il faut fabriquer l'arbre d'où une taille $O(n)$.

Le cas le pire (toutes les valeurs de la liste décroissent) conduit à un arbre de hauteur n. Le temps est alors $O(n2)$.
L'usage d'arbres AVL pallie cet inconvénient. L'algorithme est alors en $O(n \log n)$ dans le pire des cas, au détriment d'une programmation délicate.

Le tri **rapide** :
On choisit aléatoirement un indice dans la table à trier, le **pivot**.
Puis on partage les éléments de la table en ceux qui sont plus petits, qu'on place avant, et ceux qui sont plus grands, qu'on place après.
On recommence récursivement sur les deux sous listes de chaque côté du pivot.

La première fonction classe autour du pivot p les éléments de la liste et retourne l'indice où finit ce pivot.
Elle traite les éléments d'indices entre i et j. Au départ, le pivot est renvoyé en fin de tableau. Puis il est remis en place lorsque le partage est terminé.

```
Fonction entier PARTAGE(tableau t[n], entier i, j, p)
{
   g=i; d=j; echanger(t[p], t[j]) ; pivot=t[j];
   repeter
      tant que t[g] < pivot faire g++;
      tant que (d >= g) et (t[d] >= pivot) faire d--;
      si g < d alors
      {
         echanger(t[g], t[d]); g++; d--;
      }
    jusqu'a ce que g > d;
    echanger(t[g], t[j]); retour g;
}
```

On écrit maintenant la fonction principale, qui effectue récursivement le tri rapide entre les indices *i* et *j* du tableau. On suppose qu'on dispose d'une fonction choix(*i*, *j*) qui retourne un entier aléatoire entre *i* et *j*.

```
Fonction TR(tableau t[n], entier i, j)
{
   si i < j alors
   {
      p=choix(i, j);
      k=PARTAGE(t[n], i, j, p);
      TR(t[n], i, k-1); TR(t[n], k+1,j);
   }
}

Fonction TRI_RAPIDE(tableau t[n])
{
   TR(t[n], 1, n);
}
```


Pour ce qui est du temps nécessaire, celui de la fonction partage est proportionnel à $j - i$ quel que soit p.
Donc, si la liste est déjà classée et si les pivots sont successivement 1, 2, ..., n, le temps nécessaire est quadratique.

Par contre, en moyenne, le temps est en $n \log n$. Intuitivement, le pivot correspondra à une valeur centrale dans l'intervalle choisi du tableau.
Par conséquent, les intervalles sur lesquels on itère vaudront en moyenne n/2. On se retrouve alors dans une situation de type "diviser pour régner". D'où le temps moyen en $O(n \log n)$.

Remarque : comme pour les ABR, le côté aléatoire est essentiel. Si le choix du pivot n'est pas aléatoire, la complexité moyenne ne peut pas être garantie.

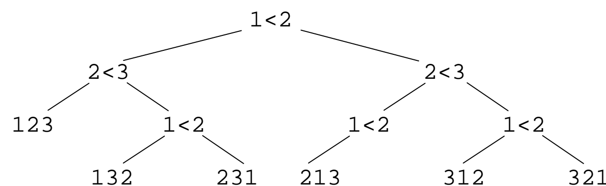
3.4 Les bornes inférieures

Dans tous les algorithmes de tri présentés, on effectue des comparaisons.
Si on trie des données numériques quelconques, on ne peut pas faire autrement.
Si on a un tel algorithme A de tri, on peut l'appliquer au cas où les listes à trier sont des **permutations**. Pour chaque permutation, on représente par un arbre binaire les comparaisons effectuées avec un branchement selon le résultat. Quand les comparaisons ont abouti à trier la permutation, on écrit celle-ci comme feuille de l'arbre.

On appelle cet arbre l'**arbre de décisions**.

Dans l'exemple suivant, on représente l'arbre de décisions pour le tri à bulles dans le cas où $n = 3$. La branche gauche signifie que le test est vrai, la branche droite qu'il est faux. L'écriture $i < j$ signifie $t[i] < t[j]$.

Les actions (échanges) ne sont pas représentées.



Pour tout algorithme de tri et toute valeur de n, on construit ainsi un arbre binaire dont les feuilles sont étiquetées par les $n!$ permutations.
Deux permutations distinctes sont à des feuilles distinctes, car on peut remonter l'arbre de décisions à partir de la liste triée 1 2 ... n.

Un arbre binaire ayant $n!$ feuilles est de hauteur au moins
$\log n! = n \log n - n \log e + (\log n)/2 + \ldots$ (Formule de Stirling)

La hauteur d'un tel arbre, qui est le nombre de comparaisons dans le pire des cas pour trier une permutation est donc minoré par un terme en $n \log n$. (On écrit $\Omega(n \log n)$). Les algorithmes de tri par fusion ou par tas sont donc optimaux dans le cas le pire.

En moyenne, la hauteur d'un arbre binaire ayant k feuilles satisfait $h_m(k) = 1 + (k_1 h_m(k_1) + k_2 h_m(k_2))/k$. (Il y a k_1 feuilles dans le sous-arbre gauche et k_2 feuilles dans le sous-arbre droit.)

Par récurrence, on peut écrire : $h_m(k) \geq 1 + (k_1 \log k_1 + k_2 \log k_2)/k$.

Cette dernière quantité est supérieure à sa valeur lorsque $k_1 = k_2 = k/2$.

Dans ce cas, on a : $h_m(k) \geq 1 + \log k - 1 = \log k$.

Pour un arbre de décisions pour les $n!$ permutations, la hauteur moyenne est donc encore au moins en $\Omega(n \log n)$.

Les tris par tas, par fusion, par ABR et le tri rapide sont donc optimaux en moyenne.