Chapitre 5

Algorithmes de tri

Le tri est une opération fondamentale de traitement des données. Il constitue le plus souvent une opération préalable à un traitement efficace de ces données. Pour se convaincre de l'utilité d'un tri, imaginez que les mots dans votre dictionnaire préféré soient mis en vrac au lieu d'être triés par ordre alphabétique et que vous ayiez à y chercher le mot "existentiel" ; la seule solution serait alors de parcourir tout le dictionnaire jusqu'à trouver le mot en question. De même, la recherche d'un nom sur une liste d'admis à un concours est grandement facilitée par un tri alphabétique de la liste des reçus. C'est ainsi que de nombreux algorithmes informatiques supposent que des données ont été préalablement triées.

Dans le cas de gros volumes de données comme on en trouve très fréquemment dans l'informatique courante, le choix d'une méthode de tri est cruciale. Il se trouve cependant qu'il n'existe pas de méthode de tri optimale universelle.

Toute méthode de tri utilise quelques opérations fondamentales comme la comparaison entre deux éléments et l'échange de deux éléments. Suivant le type de données considéré, le coût de la comparaison et celui de l'échange peuvent être très différents et, dans un souci d'efficacité, on peut être amené à privilégier un petit nombre de comparaisons ou un petit nombre d'échanges.

De plus la façon dont on accède aux données influe grandement sur le choix d'une méthode de tri :

- directement s'il s'agit de données stockées dans un tableau en mémoire centrale,
- séquentiellement s'il s'agit de données stockées dans une liste en mémoire centrale ou sur une mémoire externe (disquette ou disque dur).

Nous allons étudier ici trois algorithmes de tri de complexité quadratique :

- Le **tri par insertion** : C'est le tri du joueur de cartes. Il consiste à insérer les éléments un par un dans une partie déjà triée.
- Le **tri par sélection** : Il consiste à trouver un élément extrémal, le placer puis recommencer avec les éléments restants.
- Le **tri à bulles** : C'est une variante du tri par sélection.

Puis nous étudierons deux algorithmes de complexité améliorée à l'aide du paradigme "diviser pour régner" :

- Le **tri fusion** (Merge Sort) : Il consiste à partitionner l'entrée en deux parties de tailles similaire, les trier récursivement puis les fusionner.
- Le **tri rapide** (Quick Sort) : Il consiste à partitionner l'entrée en deux parties en comparant ses éléments à un élément pivot, puis trier ces parties récursivement.

Pour les quatre premiers tris, la série de données peut être représenté par l'une des structures de données suivantes : un tableau (type `array` en OCaml) ou une liste chaînée (type `list` en OCaml). Pour le tri rapide, la série de données sera représenté uniquement par un tableau (type `array` en OCaml).

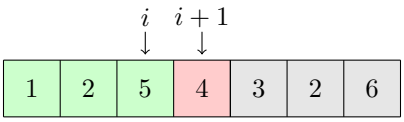
1 Tris quadratiques

1.1 Tri par insertion

Considérons une série de n données, n appartenant à $\mathbb{N}^*$.
Nous considérons en quelque sorte que les éléments à trier nous sont donnés un par un. Le premier élément à lui seul est déjà trié. Nous rangeons alors à sa bonne place le deuxième élément pour former une série triée et nous continuons ainsi, c'est-à-dire que nous insérons les éléments successivement dans une série déjà triée.

Exemple

Considérons la série de données suivantes où les i premières valeurs ont été triées :



Le tri par insertion consiste alors à insérer la $i+1$-ième valeur à la bonne place parmi les i premières valeurs :

On répète ainsi l'opération sur les valeurs restantes jusqu'à ce que l'ensemble de la série de données soit triée.

Implémentation

Le tri par insertion s'implémente récursivement de la façon suivante sur les listes :

- Si la liste est vide, elle est déjà triée;
- Sinon, on trie récursivement sa queue et on insère l'élément de tête au bon endroit dans la queue triée.

1. Écrire une fonction en OCaml `insere x l` qui insère à sa place un élément x dans une liste l déjà triée.

```
let rec insere x l = match l with
  | [] -> [x]
  | y :: _ when y > x -> x :: l
  | y :: q -> y :: (insere x q)
;;
```

2. En déduire une fonction en OCaml `tri_insertion l` qui trie une liste l d'après l'algorithme du tri par insertion.

```
let rec tri_insertion l = match l with
  | [] -> []
  | x :: q -> insere x (tri_insertion q)
;;
```

Preuve de correction

1. Pour la fonction `insere` :

   Notons pour tout $n \in \mathbb{N}$,

   $$(\mathcal{H}_n) : \text{"la fonction } \texttt{insere} \text{ est correcte pour toutes listes triées de longueur } n\text{"}.$$

   • $(\mathcal{H}_0)$ est vraie.

   • Soit $n \in \mathbb{N}^*$. Supposons que $(\mathcal{H}_{n-1})$ est vraie et montrons que $(\mathcal{H}_n)$ est aussi vérifiée.

   Soit l une liste triée de longueur n. Notons y sa tête et q sa queue. Donc y est la plus petite valeur apparaissant dans l. Si $y > x$, on insère x en tête de l et la liste obtenue est triée. Sinon, $x \geq y$. Par hypothèse de récurrence, comme q est une liste triée de longueur $n-1$, l'appel `insere x q` place x au bon endroit dans q de sorte que la liste obtenue est triée. En ajoutant ensuite y en tête, la liste obtenue est également triée car y est plus petit que x et que toutes les valeurs de q. Dans les deux cas, le résultat obtenu est correct et $(\mathcal{H}_n)$ est vraie.

   • Par le principe de récurrence, pour tout $n \in \mathbb{N}$, $(\mathcal{H}_n)$ est vraie.

2. Pour la fonction `tri_insertion` :

   Notons pour tout $n \in \mathbb{N}$,

   $$(\mathcal{H}_n) : \text{"la fonction } \texttt{tri\_insertion} \text{ est correcte pour toutes listes de longueur } n\text{"}.$$

   • $(\mathcal{H}_0)$ est vraie.

   • Soit $n \in \mathbb{N}^*$. Supposons que $(\mathcal{H}_{n-1})$ est vraie et montrons que $(\mathcal{H}_n)$ est aussi vérifiée.

   Soit l une liste de longueur n. Notons x sa tête et q sa queue. Comme q est de longueur $n-1$, l'appel `tri_insertion q` donne la liste triée des valeurs apparaissant dans q d'après l'hypothèse de récurrence. On peut donc appliquer au résultat obtenu la fonction `insere` qui place au bon endroit la tête x. On obtient alors la liste triée des valeurs apparaissant dans l. Donc $(\mathcal{H}_n)$ est vraie.

   • Par le principe de récurrence, pour tout $n \in \mathbb{N}$, $(\mathcal{H}_n)$ est vraie.

Preuve de terminaison

1. Pour la fonction `insere` :

   Soit $n \in \mathbb{N}^*$ et l une liste de longueur n. La fonction `insere x l` amène un appel récursif sur la liste q de longueur $n-1 < n$. La suite des longueurs de listes est donc une suite strictement décroissante d'entiers naturels. L'ordre sur $\mathbb{N}$ étant bien fondé, l'algorithme se termine en un nombre fini d'étapes.

2. Pour la fonction `tri_insertion` :

   On prouve que l'algorithme se termine avec les mêmes arguments que précédemment.

Complexité

1. Pour la fonction `insere` :

   La taille des données est la longueur n de la liste. Les opérations fondamentales sont les comparaisons et les ajouts en tête de liste. Notons $C_c(n)$ le nombre de comparaisons et $C_a(n)$ le nombre d'ajouts en tête de liste. Alors :

   $$\begin{cases} C_c(0) = 0 \\ \forall n \in \mathbb{N}^*,\ C_c(n) = C_c(n-1) + 1 \end{cases}$$

   Donc pour tout $n \in \mathbb{N}$, $C_c(n) = n$.

   Exactement de la même façon, on obtient pour tout $n \in \mathbb{N}$, $C_a(n) = n$.

2. Pour la fonction `tri_insertion` :

   La taille des données est encore la longueur n de la liste et les opérations fondamentales sont encore les comparaisons et les ajouts en tête de liste. Notons $T_c(n)$ le nombre de comparaisons et $T_a(n)$ le nombre d'ajouts en tête de liste. Alors :

   $$T_c(0) = 0 \qquad \text{et} \qquad \forall n \in \mathbb{N}^*,\ T_c(n) = C_c(n-1) + T_c(n-1) = (n-1) + T_c(n-1)$$

   Alors, pour tout $k \in \mathbb{N}^*$, $T_c(k) - T_c(k-1) = k-1$. En sommant pour k alors de 1 à n, on a :

   • d'une part, $\displaystyle\sum_{k=1}^{n} (T_c(k) - T_c(k-1)) = T_c(n)$ (par télescopage),

   • d'autre part, $\displaystyle\sum_{k=1}^{n} (k-1) = \frac{(n-1)n}{2}$.

   Donc, pour tout $n \in \mathbb{N}^*$, $T_c(n) = \dfrac{(n-1)n}{2} = O(n^2)$.

   Exactement de la même façon, on obtient pour tout $n \in \mathbb{N}^*$, $T_a(n) = \dfrac{(n-1)n}{2} = O(n^2)$.

On retiendra le résultat suivant :

Propriété 1 (Complexité du tri par insertion)

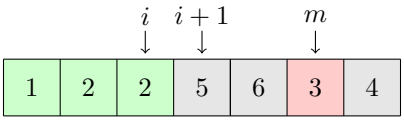
Le tri par insertion trie une série de n données avec un nombre de comparaisons et un nombre d'affectations tous les deux égaux à $O(n^2)$.

1.2 Tri par sélection

Considérons une série de n données, n appartenant à $\mathbb{N}^*$.
Le tri par sélection consiste à déterminer une valeur minimale de la série de données (cette valeur n'est peut-être pas unique), à la placer en tête et à recommencer cette opération sur les données restantes.

Exemple

Considérons la série de données suivantes où les i premières valeurs ont été triées :



Le tri par sélection consiste alors à déterminer l'indice m de la valeur minimale des données restantes puis à échanger les éléments d'indices $i+1$ et m :

On répète ainsi l'opération sur les valeurs restantes jusqu'à ce que l'ensemble de la série de données soit triée.

Implémentation

Nous allons implémenter le tri par sélection sur les tableaux.

1. Écrire une fonction en OCaml `echange t i j` qui échange le contenu des cases i et j d'un tableau t.

```
let echange t i j =
  let x = t.(i) in
    t.(i) <- t.(j);
    t.(j) <- x
;;
```

2. Écrire une fonction en OCaml `minimum t i` qui prend en argument un indice i de départ et cherche l'indice du plus petit élément dans la suite du tableau t (l'indice i inclus).

```
let minimum t i =
  let m = ref i in
    for j = i+1 to Array.length t - 1 do
      if t.(j) < t.(!m) then m := j
    done;
  !m
;;
```

3. En déduire une fonction en OCaml `tri_selection t` qui trie un tableau t d'après l'algorithme du tri par sélection.

```
let tri_selection t =
  for i = 0 to Array.length t - 2 do
    echange t i (minimum t i)
  done;
  t
;;
```

Preuve de correction

1. Pour la fonction `minimum` :

   Soit $i \in [\![0, n-2]\!]$. On pose comme invariant de boucle : pour tout $j \in [\![i+1, n-1]\!]$,

   $$(\mathcal{H}_j) : \text{"à la fin de l'itération d'indice } j,\ !m \in [\![i, j]\!] \text{ et } t.(!m) = \min\{t.(i), \dots, t.(j)\}\text{"}.$$

• L'indice $!m$ a été initialisé à i. Au premier passage dans la boucle, $j = i+1$ et on compare alors $t.(j) = t.(i+1)$ et $t.(!m) = t.(i)$. Soit $t.(j) < t.(!m)$, donc $t.(i+1) < t.(i)$ et $!m = j = i+1$. Soit $t.(j) \geq t.(!m)$, donc $t.(i+1) \geq t.(i)$ et $!m$ reste égale à i. Dans tous les cas, $!m \in [\![i, i+1]\!]$ et $t.(!m) = \min\{t.(i), t.(i+1)\}$ et $(\mathcal{H}_{i+1})$ est vraie.

• Soit $j \in [\![i+2, n-1]\!]$. Supposons $(\mathcal{H}_{j-1})$ vraie, c'est-à-dire $!m \in [\![i, j-1]\!]$ et $t.(!m) = \min\{t.(i), \ldots, t.(j-1)\}$. A l'itération d'indice j, on compare $t.(j)$ et $t.(!m)$. Soit $t.(j) < t.(!m)$, alors $!m = j$ donc $!m \in [\![i, j]\!]$ et $\min\{t.(i), \ldots, t.(j-1), t.(j)\} = t.(j) = t.(!m)$. Soit $t.(j) \geq t.(!m)$, alors $!m \in [\![i, j-1]\!] \subset [\![i, j]\!]$ et $\min\{t.(i), \ldots, t.(j-1), t.(j)\} = \min\{t.(i), \ldots, t.(j-1)\} = t.(!m)$. Dans tous les cas, $!m \in [\![i, j]\!]$ et $t.(!m) = \min\{t.(i), \ldots, t.(j)\}$ donc $(\mathcal{H}_j)$ est vraie.

• Par le principe de récurrence, pour tout $j \in [\![i+1, n-1]\!]$, $(\mathcal{H}_j)$ est vraie.

Ainsi, après la répétitive, c'est-à-dire après l'itération d'indice $n-1$, on a $!m \in [\![i, n-1]\!]$ et $t.(!m) = \min\{t.(i), \ldots, t.(n-1)\}$ donc notre algorithme est correct.

2. Pour la fonction `tri_selection` :

   On pose comme invariant de boucle : pour tout $i \in [\![0, n-2]\!]$,

   $$(\mathcal{H}_i) : \text{"à la fin de la boucle d'indice } i,\ t.(0) \leq t.(1) \leq \ldots \leq t.(i) \leq \min\{t.(i+1), \ldots, t.(n-1)\}\text{"}.$$

   • Au premier passage dans la boucle, on échange $t.(0)$ et $t.(k) = \min\{t.(0), t.(1), \ldots, t.(n-1)\}$. On a alors $t.(0) = \min\{t.(0), t.(1), \ldots, t.(n-1)\}$ donc $t.(0) \leq \min\{t.(1), \ldots, t.(n-1)\}$ et $(\mathcal{H}_0)$ est vraie.

   • Soit $i \in [\![1, n-2]\!]$. Supposons $(\mathcal{H}_{i-1})$ vraie, c'est-à-dire $t.(0) \leq \ldots \leq t.(i-1) \leq \min\{t.(i), \ldots, t.(n-1)\}$.

   A l'itération d'indice i, on échange $t.(i)$ et $t.(k) = \min\{t.(i), \ldots, t.(n-1)\}$. On a alors $t.(0) \leq t.(1) \leq \ldots \leq t.(i-1) \leq t.(i) \leq \min\{t.(i+1), \ldots, t.(n-1)\}$. Donc $(\mathcal{H}_i)$ est vraie.

   • Par le principe de récurrence, pour tout $i \in [\![0, n-2]\!]$, $(\mathcal{H}_i)$ est vraie.

   Ainsi, après la répétitive, c'est-à-dire après l'itération d'indice $n-2$, on a :

   $$t.(0) \leq t.(1) \leq \ldots \leq t.(n-2) \leq \min\{t.(n-1)\} = t.(n-1).$$

   Notre algorithme est donc correct.

Preuve de terminaison

Pour les fonctions `minimum` et `tri_selection`, nous sommes en présence de répétitives indexées qui se terminent. Donc nos algorithmes se terminent.

Complexité

1. Pour la fonction `minimum` :

   La taille des données est la longueur du tableau à partir de l'indice i, c'est à dire $n-i$. Les opérations fondamentales sont les comparaisons. Notons $C(n-i)$ le nombre de comparaisons. Alors :

   $$C(n-i) = \sum_{j=i+1}^{n-1} 1 = n-i-1.$$

2. Pour la fonction `tri_selection` :

   La taille des données est la longueur n du tableau. Les opérations fondamentales sont les comparaisons et les échanges. On note $T_c(n)$ le nombre de comparaisons et $T_e(n)$ le nombre d'échanges. Alors :

   $$\begin{aligned} T_c(n) &= \sum_{i=0}^{n-2} C(n-i) = \sum_{i=0}^{n-2} n-i-1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} = O(n^2), \\ T_e(n) &= \sum_{i=0}^{n-2} 1 = n-1 = O(n). \end{aligned}$$

On retiendra le résultat suivant :

Propriété 2 (Complexité du tri par sélection)

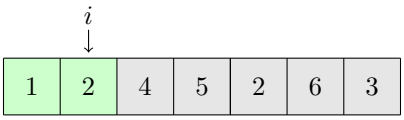
Le tri par sélection trie une série de n données avec un nombre de comparaisons égal à $O(n^2)$ et un nombre d'échanges égal à $O(n)$.

1.3 Tri à bulles

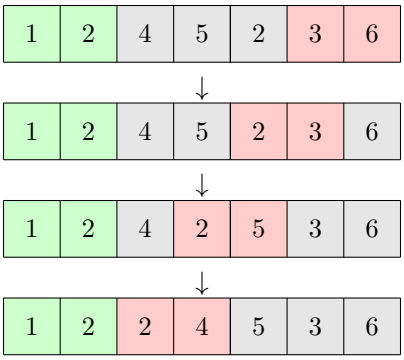
Considérons une série de n données, n appartenant à $\mathbb{N}^*$.
Le principe du tri à bulles consiste à parcourir la série de données à trier et à échanger toute inversion rencontrée. Ainsi, si l'on parcourt la série à partir de la fin, à la fin du premier passage, le minimum se retrouve en tête. Il nous reste à recommencer l'opération sur les données restantes.
Nous pouvons remarquer que si lors d'un passage, aucun échange n'est effectué, la série est alors triée.

Exemple

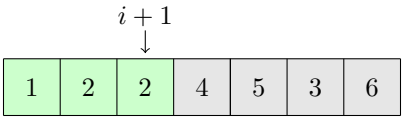
Considérons la série de données suivantes où les i premières valeurs ont été triées :



Le tri à bulle consiste à parcourir la série à partir de la fin et à échanger toute inversion rencontrée :



On obtient alors une liste dont les $i+1$ premières valeurs sont triées :



On répète ainsi l'opération sur les valeurs restantes jusqu'à ce que l'ensemble de la série de données soit triée.

Implémentation

Nous allons implémenter le tri à bulles sur les tableaux.

1. Écrire une fonction en OCaml `une_etape t i` qui réalise une étape du tri à bulles sur la partie du tableau t dont les indices sont supérieurs ou égaux à i et qui renvoie un booléen indiquant si des échanges ont été effectués.

```
let une_etape t i =
  let n = Array.length t and b = ref false in
    for k = n-1 downto i+1 do
      if (t.(k) < t.(k-1)) then
        begin
          echange t k (k-1);
          b := true
        end
    done;
    !b
;;
```

2. En déduire une fonction `tri_bulles t` qui trie le tableau t d'après l'algorithme du tri à bulles.

```
let tri_bulles t =
  let i = ref 0 in
    while (une_etape t (!i)) do
      i := !i + 1
    done;
    t
;;
```

Complexité

1. Pour la fonction `une_etape` :

   La taille des données est la longueur du tableau à partir de l'indice i, c'est-à-dire $n-i$. Les opérations fondamentales sont les comparaisons et les échanges. Notons $C_c(n-i)$ le nombre de comparaisons et $C_e(n-i)$ le nombre d'échanges. Alors :

   $$C_c(n-i) = \sum_{j=i+1}^{n-1} 1 = n-i-1 \quad \text{et de même,} \quad C_e(n-i) = n-i-1.$$

2. Pour la fonction `tri_bulles` :

   La taille des données est la longueur n du tableau. Les opérations fondamentales sont les comparaisons et les échanges. Notons $T_c(n)$ le nombre de comparaisons et $T_e(n)$ le nombre d'échanges. Alors :

   $$T_c(n) = \sum_{i=0}^{n-1} C_c(n-i) = \sum_{i=0}^{n-1}(n-i-1) = \sum_{k=0}^{n-1} k = \frac{(n-1)n}{2} = O(n^2).$$

   De même, $T_e(n) = O(n^2)$.

On en déduit le résultat suivant :

Propriété 3 (Complexité du tri à bulles)

Le tri à bulles trie une série de n données avec un nombre de comparaisons et un nombre d'échanges tous les deux égaux à $O(n^2)$.

2 Tris dichotomiques

Nous utilisons maintenant le paradigme "diviser pour régner" pour améliorer les algorithmes de tri sur les listes et les tableaux.

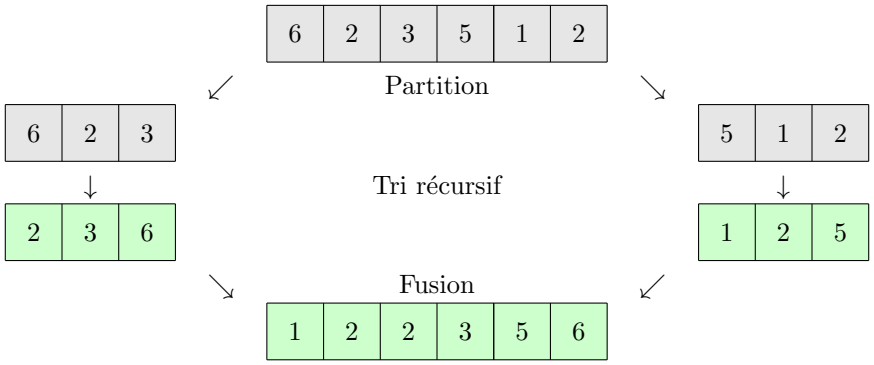
2.1 Tri fusion

Considérons une série de n données, n appartenant à $\mathbb{N}^*$.
Le tri fusion peut se décrire avec le paradigme "diviser pour régner" de la manière suivante :

- **Partition** : Nous divisons la série des n données à trier en deux séries de $n/2$ données.
- **Tri récursif** : Nous trions ensuite les deux séries récursivement.
- **Fusion** : Nous terminons en fusionnant les deux séries triées.

Exemple

Voici une illustration du tri fusion :



Implémentation

Nous allons implémenter le tri fusion sur les listes.

1. Écrire une fonction en OCaml `partition l` qui sépare une liste l en deux listes approximativement de même longueur suivant un schéma récursif. L'idée peut s'illustrer de la manière suivante : vous avez un paquet de cartes en main que vous distribuez alternativement à deux personnes.

```
let rec partition l = match l with
  | [] -> [] , []
  | [x] -> [x] , []
  | t1::t2::q -> let q1,q2 = partition q in (t1::q1 , t2::q2)
;;
```

2. Écrire une fonction en OCaml `fusion l1 l2` qui fusionne deux listes triées l_1 et l_2, non nécessairement de même longueur, en une seule liste triée suivant un schéma récursif.

```
let rec fusion l1 l2 = match (l1,l2) with
  | [] , l2 -> l2
  | l1 , [] -> l1
  | t1::q1 , t2::q2 -> if t1 <= t2 then t1::(fusion q1 l2)
                                   else t2::(fusion l1 q2)
;;
```

3. Écrire une fonction en OCaml `tri_fusion l` qui trie une liste l d'après l'algorithme du tri fusion.

```
let rec tri_fusion l = match l with
  | [] -> []
  | [x] -> [x]
  | _ -> let l1,l2 = partition l in
         fusion (tri_fusion l1) (tri_fusion l2)
;;
```

Complexité

Les opérations fondamentales sont les comparaisons.

1. Pour la fonction `partition` :

   Cette fonction ne nécessite aucune comparaison.

2. Pour la fonction `fusion` :

   La taille des données est la somme des longueurs $n_1 + n_2$ des deux listes l_1 et l_2. Cette fonction nécessite au plus $n_1 + n_2 - 1 = O(n_1 + n_2)$ comparaisons.

3. Pour la fonction `tri_fusion` :

   La taille des données est la longueur n de la liste l. Notons $T(n)$ le nombre de comparaisons. On a alors :

$$T(n) = \underbrace{0}_{\text{partition}} + \underbrace{2T(n/2)}_{\text{appels récursifs}} + \underbrace{O(n)}_{\text{fusion}}$$

   En appliquant le théorème maître de complexité, avec $q = 2$ et $\gamma = 1$, on obtient $T(n) = O(n \log_2(n))$.

On retiendra le résultat suivant :

Propriété 4 (Complexité du tri fusion)

Le tri fusion trie une série de n données avec un nombre de comparaisons égal à $O(n \log_2(n))$.

2.2 Tri rapide

Considérons une série de n données, n appartenant à $\mathbb{N}^*$.
Le tri rapide peut se décrire de nouveau avec le paradigme "diviser pour régner", mais l'idée de partitionnement est différente :

- **Partition** : Nous choisissons un élément p appartenant à la série de données. Le partitionnement consiste à réorganiser la série de façon à obtenir la configuration suivante :

éléments $\leq p$	p	éléments $> p$

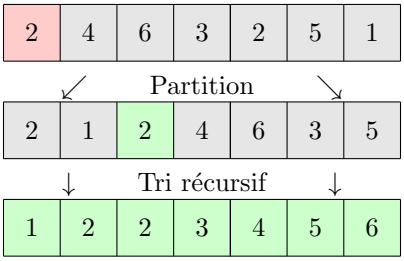
  Nous pouvons ainsi assurer que l'élément p se trouve à sa place.

- **Tri récursif** : Nous trions ensuite récursivement les deux portions de la série (les éléments inférieurs ou égaux à p sauf p et les éléments strictement supérieurs à p) selon le même principe.

- **Fusion** : Lorsque ces deux portions sont triées, nous pouvons affirmer que la série est triée en totalité (pas de fusion ici).

Exemple

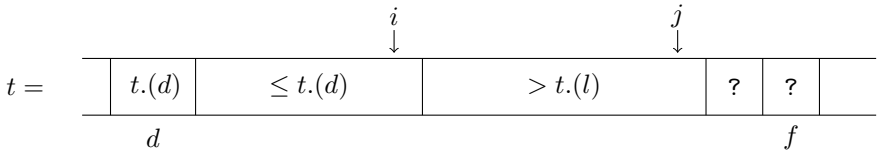
Voici une illustration du tri rapide (en prenant comme pivot p le premier élément de la série de données) :



Implémentation

Nous allons implémenter le tri rapide sur les tableaux.

1. Écrire une fonction en OCaml `partition t d f` qui réorganise le sous-tableau $[\![t.(d); t.(d+1); \ldots; t.(f)]\!]$ suivant la méthode décrite ci-dessus, le pivot p choisi étant $t.(d)$. Nous respecterons l'invariant de boucle suivant :



   Nous terminerons en échangeant les valeurs de $t.(d)$ et $t.(i)$.

```
let partition t d f =
  let p = t.(d) and i = ref d in
    for j = d+1 to f do
      if t.(j) <= p then
        begin
          i := !i + 1;
          echange t !i j
        end
    done;
    echange t d !i;
    !i
;;
```

2. Écrire une fonction en OCaml `tri_rapide t` qui trie le tableau t d'après l'algorithme du tri rapide.

```
let tri_rapide t =
  let rec aux t d f =
    if f > d then
      begin
        let m = partition t d f in
          aux t d (m-1);
          aux t (m+1) f
      end
  in
  aux t 0 (Array.length t - 1)
;;
```

Complexité

1. Pour la fonction `partition` :

   La taille des données est la longueur du tableau entre les indices d et f, c'est-à-dire $f-d+1$. Les opérations fondamentales sont les comparaisons et les échanges. Notons $C(f-d+1)$ le nombre de comparaisons et d'échanges. Alors :

$$C(f-d+1) = (f-d) + (f-d+1) = 2(f-d)+1.$$

2. Pour la fonction `tri_rapide` :

   La taille des données est la longueur n du tableau. Les opérations fondamentales sont les comparaisons et les échanges. Notons $T(n)$ le nombre de comparaisons et d'échanges. On a alors :

$$T(n) = \underbrace{2(n-1)+1}_{\text{partition}} + \underbrace{T(m)+T(n-m-1)}_{\text{appels récursifs}} + \underbrace{0}_{\text{fusion}}.$$

   Donnons plusieurs interprétations pour analyser cette équation de complexité :

(a) Dans le meilleur des cas :
On espère que le choix aléatoire du premier élément du tableau comme pivot a permis de diviser le tableau à trier de taille n en deux sous-tableaux à peu près de même taille $n/2$. Cela nous amène à une équation de complexité de la forme :

$$T(n) = 2T(n/2) + O(n).$$

En appliquant le théorème maître de complexité, avec $q = 2$ et $\gamma = 1$, on obtient $T(n) = O(n \log_2(n))$.

(b) Dans le pire des cas :
Dans le pire des cas (si le tableau est trié dans l'ordre décroissant), le choix aléatoire du premier élément du tableau comme pivot amène une partition déséquilibrée du tableau de taille n en un sous-tableau de taille $n-1$ et un sous-tableau vide. Cela nous amène à une équation de complexité de la forme :

$$T(n) = \underbrace{2n+1}_{\text{partition}} + \underbrace{T(n-1)}_{\text{appel récursif}}.$$

Donc : $\forall k \in \mathbb{N}^*$, $T(k) - T(k-1) = 2k+1$. Par somme :

$$T(n) = \sum_{k=1}^{n}(T(k) - T(k-1)) = \sum_{k=1}^{n}(2k+1) = 2\sum_{k=1}^{n} k + \sum_{k=1}^{n} 1 = 2\frac{n(n+1)}{2} + n = n(n+2).$$

Finalement, $T(n) = O(n^2)$.

(c) En moyenne :
Nous choisissons arbitrairement le premier élément du tableau comme pivot. Nous supposons que la partition du tableau selon le pivot suit une loi de probabilité uniforme. Cela nous amène à une équation de complexité de la forme :

$$\begin{aligned} T(n) &= \underbrace{2n+1}_{\text{partition}} + \sum_{k=0}^{n-1} \frac{1}{n} \underbrace{(T(k) + T(n-k-1))}_{\text{pivot en position } k} \\ &= 2n+1 + \frac{1}{n}\left(\sum_{k=0}^{n-1} T(k) + \sum_{k=0}^{n-1} T(n-k-1)\right) \\ &= 2n+1 + \frac{2}{n}\sum_{k=0}^{n-1} T(k). \end{aligned}$$

Comme $T(n-1) = 2n-1 + \frac{2}{n-1}\sum_{k=0}^{n-2} T(k)$ et donc $\sum_{k=0}^{n-2} T(k) = \frac{n-1}{2}(T(n-1) - (2n-1))$, on a :

$$\begin{aligned} T(n) &= 2n+1 + \frac{2}{n}\sum_{k=0}^{n-1} T(k) \\ &= 2n+1 + \frac{2}{n}\left(\sum_{k=0}^{n-2} T(k) + T(n-1)\right) \\ &= 2n+1 + \frac{2}{n}\left(\frac{n-1}{2}(T(n-1) - (2n-1)) + T(n-1)\right) \\ &= 2n+1 + \frac{n+1}{n}T(n-1) - \frac{(n-1)(2n-1)}{n} \\ &= \frac{n+1}{n}T(n-1) + \frac{4n-1}{n}. \end{aligned}$$

En divisant par $n+1$, on obtient la relation :

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{4n-1}{n(n+1)}.$$

En posant $u_k = \frac{T(k)}{k+1}$, on a alors :

$$\forall k \in \mathbb{N},\ u_k - u_{k-1} = \frac{4k-1}{k(k+1)}.$$

Par somme, $\sum_{k=1}^{n}(u_k - u_{k-1}) = \sum_{k=1}^{n}\frac{4k-1}{k(k+1)}$. D'une part, $u_n = \sum_{k=1}^{n}(u_k - u_{k-1})$ (par télescopage). D'autre part,

$$\begin{aligned}\sum_{k=1}^{n}\frac{4k-1}{k(k+1)} &= \sum_{k=1}^{n}\left(-\frac{1}{k}+\frac{5}{k+1}\right) = -\sum_{k=1}^{n}\frac{1}{k}+5\sum_{k=1}^{n}\frac{1}{k+1} = -\sum_{k=1}^{n}\frac{1}{k}+5\sum_{k=2}^{n+1}\frac{1}{k}\\ &= -1-\sum_{k=2}^{n}\frac{1}{k}+5\sum_{k=2}^{n}\frac{1}{k}+\frac{5}{n+1} = 4\sum_{k=2}^{n}\frac{1}{k}+\frac{5}{n+1}-1.\end{aligned}$$

Donc $u_n = 4\sum_{k=2}^{n}\frac{1}{k}+\frac{5}{n+1}-1$.

Pour obtenir un équivalent de u_n, on fait une comparaison somme/intégrale :

$$\forall k \in [\![2,n]\!],\ \int_k^{k+1}\frac{dx}{x} \leq \frac{1}{k} \leq \int_{k-1}^{k}\frac{dx}{x},$$

donc, en sommant,

$$\underbrace{\int_2^{n+1}\frac{dx}{x}}_{=\ln(n+1)-\ln(2)} \leq \sum_{k=2}^{n}\frac{1}{k} \leq \underbrace{\int_1^{n}\frac{dx}{x}}_{=\ln(n)}.$$

Ainsi, $\sum_{k=2}^{n}\frac{1}{k} \sim \ln(n)$ donc $u_n \sim 4\ln(n)$ et $T(n) \sim 4n\ln(n)$.

Finalement, $T(n) = O(n\log_2(n))$.

On en déduit le résultat suivant :

Propriété 5 (Complexité du tri rapide)

Le tri rapide trie une série de n données :

- en $O(n\log_2(n))$ comparaisons et échanges dans le meilleur des cas,
- en $O(n^2)$ comparaisons et échanges dans le pire des cas,
- en $O(n\log_2(n))$ comparaisons et échanges en moyenne.

Remarque. Nous retiendrons les temps de partition et de fusion des deux tris dichotomiques :

	Partition	Fusion
Tri fusion	0	linéaire
Tri rapide	linéaire	0