Corrigé TD sur les tris

1. On parcourt le tableau en mettant à jour deux références indiquant les indices des deux plus grands éléments jusqu'à la position k.

```
let deuxieme t =
  let n = Array.length t and first = ref 0 and second = ref 0 in
  if t.(0) > t.(1) then second := 1 else first := 1;
  for k = 2 to Array.length t -1 do
    if t.(k) > t.(!first) then (second := !first; first := k)
    else if t.(k) > t.(!second) then second := k
  done;
  !second ;;
```

   On compte au plus $1 + 2(n-2) = 2n-3$ comparaisons.

2. (a) On procède récursivement en parcourant le tableau entre les indices 0 et i.

```
let retourne t i =
  let rec aux i j =
   if i <= j then
     begin
     let k = t.(i) in
     t.(i) <- t.(j);
     t.(j) <- k;
     aux (i+1) (j-1)
    end
  in aux 0 i ;;
```

   Le nombre d'affectations est en $\Theta(i)$.

   (b) Pour amener le plus grand élément en dernière position (c'est-à-dire $n-1$), on procède ainsi :
   - on cherche l'indice i du plus grand élément.
   - on retourne le tableau jusqu'au rang i.
   - on retourne le tableau jusqu'au rang $n-1$.

   (c) On trie le tableau en plaçant successivement les plus grands éléments : le plus grand, puis le second, et ainsi de suite.

```
let tri t =
  let n = Array.length t in
  for k = n-1 downto 1 do
    let m = ref t.(0) in
    let i = ref 0 in
    for j = 1 to k do
      if t.(j) > !m then (m := t.(j); i := j)
    done;
    retourne t (!i);
    retourne t k
  done ;;
```

   Comme chaque appel à `retourne` est linéaire, l'algorithme est quadratique.

3. (a) On filtre selon les cas d'inversion possible dans les deux premières couleurs.

```
let rec modification l = match l with
  | [] -> [], false
```

```
  | Blanc::Rouge::q -> let ll,b = modification (Blanc::q) in Rouge::ll, true
  | Bleu::Rouge::q -> let ll,b = modification (Bleu::q) in Rouge::ll, true
  | Bleu::Blanc::q -> let ll,b = modification (Bleu::q) in Blanc::ll, true
  | t::q -> let ll,b = modification q in t::ll, b  ;;
```

(b) On discute selon que la fonction précédente a modifié ou non la liste.

```
let rec tri_drapeau l =
  let ll,b = modification l in
  if b then tri_drapeau ll else l ;;
```

4. (a) Supposons $m < m'$. Alors $\text{card}\{x \in E \,|\, x < m\} + 1 \leqslant \text{card}\{x \in E \,|\, x < m'\}$ d'où l'unicité de la médiane. L'existence est évidente car il suffit de classer les éléments dans l'ordre et de prendre le numéro $\lfloor \frac{n}{2} \rfloor$.

   (b) On parcourt le tableau et, pour chaque élément, on compte le nombre d'éléments qui lui sont strictement inférieurs. On renvoie l'élément pour lequel ce nombre est $\lfloor \frac{n}{2} \rfloor$.

```
let mediane t =
  let n = Array.length t and i = ref 0 and res = ref 0 and compte = ref 0 in
  while !i < n do
    for k = 0 to n-1 do
     if t.(k) < t.(!i) then incr compte;
  done;
  if !compte = n/2 then res := !i else compte := 0;
       incr i
  done;
  t.(!res) ;;
```

   (c) On utilise la méthode diviser pour régner.

```
let rec mediane2 u v = match Array.length u with
  | 1 -> max u.(0) v.(0)
  | n when n mod 2 = 1 -> if u.(n/2) > v.(n/2) then
    mediane2 (Array.sub u 0 (n/2+1)) (Array.sub v (n/2) (n/2+1))
    else
    mediane2 (Array.sub v 0 (n/2+1)) (Array.sub u (n/2) (n/2+1))
  | n -> if u.(n/2-1) > v.(n/2-1) then
    mediane2 (Array.sub u 0 (n/2)) (Array.sub v (n/2) (n/2))
    else
     mediane2 (Array.sub v 0 (n/2)) (Array.sub u (n/2) (n/2)) ;;
```

Le nombre de comparaisons vérifie l'équation de complexité $u_n = u_{\lfloor \frac{n}{2} \rfloor} + u_{\lceil \frac{n}{2} \rceil} + 1$ d'où $u_n = \Theta(n)$.

5. (a) La méthode "bourrine" consiste à trier le tableau et à renvoyer le k-ième élément. On passe par le tri fusion des listes pour assurer une complexité en $n \log n$ via des fonctions de conversion de liste en vecteur et vice-versa.

```
let rec decoupe = function
   | [] -> [], []
   | [a] -> [a], []
   | a::b::q -> let l,m = decoupe q in (a::l), (b::m)
;;
let rec fusion l m = match (l,m) with
   | ([],m) -> m
   | (l,[]) -> l
   | (a::q, b::r) when a < b -> a::(fusion q m)
   | (a::q, b::r)            -> b::(fusion l r)
;;
let rec tri_fusion = function
   | [] -> []
   | [a] -> [a]
   | l -> let (l1,l2) = decoupe l in fusion (tri_fusion l1) (tri_fusion l2)
```

```
;;
let select1 k t =
  let l = tri_fusion (Array.to_list t) in
  (Array.of_list l).(k-1)
;;
```

(b) Si les $k-1$ premiers éléments d'un tableau sont rangés dans l'ordre en tête, on trouve le k-ième en $n-k+1$ comparaisons, d'où l'algorithme suivant (codé en récursif) où `aux k` trie les k premiers éléments de t suivant un tri sélection. La complexité est bien en $\mathcal{O}(kn)$.

```
let select2 k t =
 let rec aux = function (* aux k trie les k premiers éléments du tableau *)
     0 -> ()
   | j -> aux (j-1);
       let ind = ref (j-1) in
       for i = j to Array.length t -1 do
        if t.(i) < t.(!ind) then ind := i
       done;
       let temp = t.(j-1) in (t.(j-1) <- t.(!ind); t.(!ind) <- temp)
       in aux k;
       t.(k-1)  ;;
```

(c)
```
let echange i j t = let aux = t.(i) in t.(i) <- t.(j); t.(j) <- aux
;;
let partition t i j =
  let g = ref(i+1) and d = ref j in
  while !d >= !g do
    if t.(!g)  <= t.(i) then incr g else (echange !g !d t; decr d)
  done;
  if !d > i then echange i !d t;
  !d -i ;;
```

(d) On partitionne la zone du tableau étudiée par rapport au premier élément de cette zone, on récupère la position et on exécute la sélection sur une des deux parties du tableau (alors que dans le tri rapide, on doit trier récursivement les deux parties).

```
let select k t =
  let rec select_aux k i j =
   let p = partition t i j in
   if k = p+1 then t.(i+p)
   else (if k <= p then select_aux k i (i+p-1)
          else select_aux (k-p-1) (i+p+1) j)
  in select_aux k 0 (Array.length t -1) ;;
```

En notant u_n le nombre de comparaisons effectuées en moyenne, on a $u_n = n-1+\frac{1}{n}\sum_{k=0}^{n-1} u_k$. On en déduit facilement (avec $u_1 = 0$) que $u_n = u_{n-1} + \frac{2(n-1)}{n}$. Par suite, $u_n - u_{n-1} \xrightarrow[n\to\infty]{} 2$, d'où par théorème de Cesaro, $u_n \underset{n\to\infty}{\sim} 2n$.

Comme dans le tri rapide, le pire des cas correspond à la situation où à chaque partition, on perd un seul élément (par exemple lorsque le tableau est déjà trié), la complexité est alors de l'ordre de $k\,n$.