1 Tri par sélection

1.

```
let rec min_liste = function
   []   -> failwith "liste vide"
  |[a]  -> (a,[])
  |a::q -> let (b,l) = min_liste q in
           if a<b then (a,q)
           else (b,a::l) ;;
```

2.

```
let rec tri_selection = function
  [] -> []
 |l  -> let (a,q) = min_liste l in
        a::(tri_selection q) ;;
```

3. On commence par calculer le complexité de la fonction `min_liste`. Si on note $M(n)$ cette complexité pour une liste de longueur n, on a, en comptant le nombre de comparaisons :

$$\begin{aligned} M(1) &= 0 \\ M(n) &= M(n-1)+1 \quad \text{si } n > 1 \end{aligned}$$

D'où $M(n) = n-1$ pour $n \geq 1$.
Soit $T(n)$ la complexité de la fonction `tri_selection` pour une liste de taille n. T vérifie :

$$\begin{aligned} T(0) &= 0 \\ T(n) &= M(n) + T(n-1) \quad \text{si } n \geq 1 \end{aligned}$$

On a ainsi $T(k) - T(k-1) = M(k) = k-1$ pour tout $k \geq 1$ et par conséquent

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

Donc $T(n) = \frac{n(n-1)}{2} = \mathcal{O}(n^2)$. Il n'y a pas de raison de distinguer ici la complexité en moyenne et la complexité dans le pire des cas puisque le nombre de comparaisons effectuées dans ce tri ne dépend que de la taille de la liste à trier et pas de la valeur de ses éléments.

2 Tri par tas

4. D'après la définition d'un tas le minimum est égal à la racine.

```
let min_tas = function
  Vide -> failwith "tas vide"
 |Noeud (a,_,_) -> a ;;
```

5.

```
let rec enleve_min = function
  Vide -> Vide
 |Noeud (_,Vide,d) -> d
 |Noeud (_,g,Vide) -> g
 |Noeud (_,g,d) -> let min_g = min_tas g and min_d = min_tas d in
                   if min_g < min_d then Noeud (min_g,enleve_min g,d)
                                    else Noeud (min_d,g,enleve_min d) ;;
```

6.

```
let rec ajouter x = function
  Vide -> Noeud (x,Vide,Vide)
 |Noeud (a,g,d) -> if a<x then Noeud (a,d,ajouter x g)
                          else Noeud (x,d,ajouter a g) ;;
```

On peut prouver que cette fonction permet de maintenir un arbre avec une hauteur h vérifiant

$$h \leq \ln_2 n$$

7.

```
let rec ajouter_liste = function
  [] -> Vide
 |a::q -> ajouter a (ajouter_list q) ;;
```

8.

```
let rec vider = function
  Vide -> []
 |Noeud (a,g,d) as t -> a::(vider (enleve_min t)) ;;
```

9.

```
let tri_par_tas l = vider (ajouter_liste l) ;;
```

10. Dans le pire des cas la fonction `ajouter` effectue h comparaisons pour un arbre de hauteur h.
Si l est une liste de taille n la fonction `ajouter_liste` appliquée à l effectue n appels à `ajouter` sur des arbres comportant respectivement $0, 1, \ldots, n-1$ nœuds. D'après la propriété de la fonction `ajouter`, chacun de ces arbres à une hauteur bornée par

$$0, \ln_2(1), \ldots, \ln_2(n-1)$$

On en déduit que dans le pire des cas la complexité de `ajouter_liste` pour une liste de taille n est bornée (grossièrement) par :

$$0 + \ln_2(1) + \cdots + \ln_2(n-1) \leq n \ln_2 n$$

Dans la fonction `tri_par_tas`, `(ajouter_liste l)` est un tas comportant n nœuds (si l est de longueur n) et de hauteur $h \leq \ln_2 n$. La fonction `vider` va effectuer n appel

à `enleve_min` sur des arbres de plus en petits $a_1, \ldots, a_n$ dont la hauteur sera toujours inférieur à h, donc à $\ln_2 n$.
Or dans le pire des cas, la fonction `min_liste` effectue h_a comparaisons pour un arbre a de hauteur h_a. Dans notre cas le nombre de comparaisons effectuées par la fonction `vider` sera donc borné par :

$$h_{a_1} + \cdots + h_{a_n} \leq h + \cdots + h \leq n \ln_2 n$$

On en déduit que la complexité dans le pire des cas de `tri_par_tas` est bornée par

$$2n \ln_2 n = \mathcal{O}(n \ln n)$$

3 Tri par tas dans un tableau

11.

```
let max_noeud t i =
  let m = ref i in
  if 2*i+1 < t.taille && t.tab.(i) < t.tab.(2*i+1) then
    m := 2*i+1;
  if 2*i+2 < t.taille && t.tab.(!m) < t.tab.(2*i+2) then
    m := 2*i+2;
  !m ;;
```

Avant de lire la valeur d'un fils d'un nœud, il faut s'assurer que ce fils existe. Pour cela, on teste si son indice est valide, c'est à dire si il est bien inférieur strict à la taille courante du tas.
On utilise ici une référence pour la variable m car elle est peut être modifiée dans le programme.

12.

```
let rec entasser t i =
  let m = max_noeud t i in
  if m<>i then (
    let temp=t.tab.(i) in
    t.tab.(i) <- t.tab.(m);
    t.tab.(m) <- temp;
    entasser t m ) ;;
```

Si $m = i$ le nœud d'indice i est déjà plus grand que ses fils donc le sous-arbre enraciné en i est déjà un tas. Dans le cas contraire, on intervertit le nœud i est son plus grand fils. Il faut alors faire un appel récursif au niveau du sous-arbre modifié qui n'est plus forcément un tas.

13.

```
let construire_tas tab =
  let t = { tab=tab; taille=(vect_length tab) } in
  for i=t.taille/2-1 downto 0 do
    entasser t i
  done;
  t ;;
```

Chaque feuille représente déjà un tas valide, donc on commence à appeler la fonction `entasser` à partir des sous-arbres qui ne sont pas réduits à une feuille. On parcoure le tableau de droite à gauche pour avoir toujours la propriété que les fils du nœud courant sont déjà des tas.

14.

```
let retire_max t =
  let temp = t.tab.(0) in
  t.tab.(0) <- t.tab.(t.taille-1);
  t.tab.(t.taille-1) <- temp;
  t.taille <- t.taille -1 ;
  entasser t 0 ;;
```

On échange la racine de l'arbre avec le dernier élément valide du tableau et on décrémente la taille du tas. Ce dernier contient donc un élément de moins. La valeur de la racine du nouveau tas n'étant certainement pas valide il faut faire un appel à `entasser`. Cet appel est cohérent car les fils de la racine sont bien des tas.

15.

```
let vide_tas t =
  while t.taille > 1 do
    retire_max t
  done ;;
```

Au fur et à mesure que l'on vide le tas, la zone droite du tableau qui contient les éléments non valide du tas (d'indice supérieur ou égale à la taille courante) se remplie avec les plus grands éléments du tas initial trié par ordre croissant. Lorsque le tas n'a plus qu'un élément, le tableau est trié.

16.

```
let tri_par_tas_vect tab =
  let t = construire_tas tab in
  vide_tas t;
  t.tab ;;
```