I CCP 2023 : sélection du $(k+1)^e$ plus petit élément

1.

```
let rec longueur = function
    | [] -> 0
    | _::l -> 1 + longueur l
```

2.

```
let rec insertion l a = match l with
    | [] -> [a]
    | b::l -> if a <= b then a::b::l else b::insertion l a
```

3.

```
let rec tri_insertion = function
    | [] -> []
    | a::l -> insertion (tri_insertion l) a
```

4.

```
let selection_n l n =
    let rec aux l n = match l with
        | [] -> failwith "Pas assez d'éléments"
        | a::l -> if n = 0 then a else aux l (n - 1)
    in aux (tri_insertion l) n
```

5.

```
let cinq l = (* renvoie (5 premiers éléments de l, reste de l) *)
    let rec aux l n = match l with
        | [] -> [], []
        | a::q -> if n = 0 then [], l else
            let l1, l2 = aux q (n - 1) in
            a::l1, l2
    in aux l 5

let rec paquets_de_cinq l =
    let l1, l2 = cinq l in
    if l1 = [] then [] else l1::paquets_de_cinq l2
```

6.

```
let median l =
    selection_n l (longueur l / 2)

let rec medians = function
    | [] -> []
    | l::ll -> median l::medians ll
```

7.

```
let rec partage e l = match l with
    | [] -> [], [], 0, 0
    | a::l -> let l1, l2, n1, n2 = partage e l in
        if a <= e then a::l1, l2, 1 + n1, n2
        else l1, a::l2, n1, 1 + n2
```

8.

```
let selection l k =
    let n = longueur l in
    if n <= 5 then selection_n l k
    else
        let l_cinq = paquets_de_cinq l in
        let pivot = median (medians l_cinq) in
        let l1, l2, n1, n2 = partage pivot l in
        if k < n1 then selection l1 k
        else selection l2 (k - n1)
```

II Centrale-Supélec 2023

11. On parcourt la liste `l` en complexité linéaire en la taille de `l` :

```
let rec succ_list l e = match l with
    | [] -> -1
    | t::q -> if t > e then t else succ_list q e
```

12.
    - déterminer le maximum : $O(1)$ en renvoyant l'élément d'indice n
    - tester l'appartenance : $O(\ln n)$ par recherche dichotomique parmi les n premiers éléments
    - ajouter un élément : il faut incrémenter la première case du tableau, trouver où insérer l'élément puis décaler les éléments suivants, en $O(n)$.

13.

```
let succ_vect t x =
    let n = Array.length t in
    if x >= t.(n - 1) then -1 (* pas de successeur *)
    else
        let i, j = ref 0, ref (n - 1) in
        while !i < !j do
            let m = (!i + !j) / 2 in
            if t.(m) <= x then i := m + 1 (* le successeur de x doit être à droite *)
            else j := m (* à gauche *)
        done;
        !i
```

14. Soit $C(n)$ la complexité de `succ_vect` pour un ensemble de taille n. On a $C(1) = 1$ et $C(n) = C(n/2) + O(1)$ pour $n \geqslant 2$. On a donc $C(n) = O(\ln(n))$.

15. On peut parcourir `t1` et `t2` avec des indices `i` et `j` et ajouter à chaque fois le plus petit à `t` :

```
let union_vect t1 t2 =
  let n1, n2 = t1.(0), t2.(0) in
  let t = Array.make (Array.length t1) 0 in
  t.(0) <- n1 + n2;
  let i, j, k = ref 1, ref 1, ref 1 in
  while !i < n1 && !j < n2 do
      if t1.(!i) < t2.(!j) then begin
          t.(!k) <- t1.(!i);
          incr i
      end else if t1.(!i) > t2.(!j) then begin
          t.(!k) <- t2.(!j);
          incr j
      end else begin
          t.(!k) <- t1.(!i);
          incr i;
          incr j
      end;
      incr k
  done;
  while !i < n1 do
      t.(!k) <- t1.(!i);
      incr i;
      incr k
  done;
  while !j < n2 do
      t.(!k) <- t2.(!j);
      incr j;
      incr k
  done;
  t;;
```

16. On regarde toujours à gauche :

```
type abr = Nil | Noeud of int * abr * abr

let rec min_abr = function
    | Nil -> -1
    | Noeud (x, Nil, _) -> x
    | Noeud (_, g, _) -> min_abr g
```

17. On suppose dans le code suivant qu'il n'y a pas de doublon (x apparaît au plus une fois). Sinon, on peut supprimer les occurences multiples de x.

```
let rec partitionne_abr x = function
    | Nil -> false, Nil, Nil
    | Noeud(r, g, d) ->
       if x = r then true, g, d
       else if x < r then
           let b, g1, d1 = partitionne_abr x g in
           b, g1, Noeud(r, d1, d)
       else
           let b, g1, d1 = partitionne_abr x d in
           b, Noeud(r, g, g1), d1
```

18. On peut réutiliser la fonction précédente :

```
let rec insertion_abr a x =
    let b, g, d = partitionne_abr x a in
    Noeud(x, g, d)
```

On peut aussi utiliser la méthode classique :

```ocaml
let rec insertion_abr a x = match a with
    | Nil -> Noeud(x, Nil, Nil)
    | Noeud(r, g, d) ->
        if x < r then Noeud(r, insertion_abr g x, d)
        else Noeud(r, g, insertion_abr d x)
```

19. On peut ajouter tous les éléments de `a1` à `a2` avec `insertion_abr`, sachant qu'on peut permuter les `union_abr` :

```ocaml
let rec union_abr a1 a2 = match a1 with
  | Nil -> a2
  | Noeud(r, g, d) -> union_abr g (union_abr d (insertion_abr a2 r))
```

20. Remarquons d'abord qu'il y a 2^k noeuds à profondeur k dans un arbre binaire complet (on peut le démontrer par récurrence). Il y a $\sum_{i=0}^{k-1} 2^i = 2^k - 1$ noeuds de profondeur inférieure à k. Donc un noeud de profondeur k a un numéro entre 2^k et $2^{k+1} - 1$.
Soit k la profondeur du noeud i. Alors $2^k \leqslant i \leq 2^{k+1} < 2^{k+1}$. Donc $k = \lfloor \log_2 i \rfloor$.
Le sous-arbre enraciné en i est un arbre binaire complet de hauteur $p - k$ donc possède 2^{p-k} feuilles (à profondeur $p - k$).

21. Les fils de i sont $2i$ et $2i + 1$. Son père est $\left\lfloor \frac{i}{2} \right\rfloor$.

22. On remplie les feuilles puis les autres noeuds récursivement :

```ocaml
let fabrique l n =
  let e = Array.make (2*n) false in
  List.iter (fun x -> e.(n + x) <- true) l;
  let rec aux i =
    if i < n then e.(i) <- aux (2*i) || aux (2*i + 1);
    e.(i) in
  aux 1;
  e
```

23. On met `true` à toutes les positions correspondantes dans l'arbre. Dans le meilleur cas, l'élément était déjà dans E et on ne modifie rien.

```ocaml
let insere e k =
  let n = Array.length e in
  if not e.(n/2 + k) then (
    let rec aux i =
      if i > 0 && not e.(i) then (
        e.(i) <- true;
        aux (i/2)
      ) in
    aux (n/2 + k))
```