1 Analyse de Programme

Exercice 1 :

1. Les valeurs prises par la variable `s` forment une suite strictement croissante à valeurs dans $\mathbb{N}$, la condition `!s <= n` ne peut donc rester vraie indéfiniment. Ainsi, la boucle `while` (et donc la fonction `f`) termine.
2. Conjecture : `f n` $= \lfloor\sqrt{\texttt{n}}\rfloor$. On prend cette relation comme spécification de `f`.
3. Montrons que la propriété suivante est un invariant de boucle :

$$\texttt{s} = (\texttt{c}+1)^2 \tag{1}$$

   - Avant la boucle, on a `c` $= 0$ et `s` $= 1 = (\texttt{c}+1)^2$.
   - Notons c et s les valeurs des variables en haut du corps de la boucle, et c' et s' leurs valeurs en bas du corps de la boucle. Supposons que $s = (c+1)^2$, et montrons que $s' = (c'+1)^2$. On a $c' = c+1$ (ligne 5), et après la ligne 6 on a :

$$s' = \underbrace{s}_{(c+1)^2} + 2c' + 1 = (\underbrace{c+1}_{c'})^2 + 2c' + 1 = c'^2 + 2c' + 1 = (c'+1)^2$$

   Ainsi, la propriété (1) est bien un invariant de boucle.
   Lorsqu'on sort de la boucle `while`, on a donc $(c+1)^2 = s > n$. De plus, la valeur précédente de `s` (qui vaut c^2) respectait la condition du `while`, donc $c^2 \leq n$. Au final, on a $c^2 \leq n \leq (c+1)^2$.
   Comme $x \mapsto \sqrt{x}$ est croissante, on a : $c \leq \sqrt{n} < c+1$, donc `f` renvoie $c = \lfloor\sqrt{n}\rfloor$.
   La fonction `f` est donc correcte.
4. On passe $c = \lfloor\sqrt{n}\rfloor$ fois dans la boucle while, et on effectue un nombre constant d'opérations élémentaires à chaque passage dans la boucle. On effectue de plus un nombre constant d'opérations élémentaires en dehors de la boucle. La complexité de `f` est donc un $O(\sqrt{n})$.

2 Valeur majoritaire

Exercice 2 :

1.
```
let occurrences t =
  (* Renvoie un tableau occ tel que pour tout i
     occ.(i) est le nombre d'occurrences de t.(i) dans t *)
  let n = Array.length t in
  let occ = Array.make n 0 in
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if t.(i) = t.(j) then
        occ.(i) <- occ.(i) + 1
    done ;
  done ;
  occ
;;

let indice_max t =
  (* calcule l'indice du max du tableau t *)
  let n = Array.length t in
  let imax = ref 0 in (* imax est l'indice du max du tableau *)
  for i = 1 to n-1 do
    if t.(i) > t.(!imax) then imax := i
  done ;
  !imax
;;

let valeur_majoritaire t =
  let imax = indice_max (occurrences t) in
  t.(imax)
;;
```

2. Toutes les fonctions utilisées ne sont pas récursives et n'utilisent pas de boucle `while`, donc la fonction `valeur_majoritaire` termine.

3. Notons n la longueur de `t`. La fonction `occurrences` parcourt les éléments de `t` à l'aide de deux boucles `for` imbriquées, sa complexité est donc en $O(n^2)$. La fonction `indice_max` a une complexité linéaire, et est utilisée sur un tableau de taille n. Au final, la fonction `valeur_majoritaire` a donc une complexité en $O(n^2)$.

4. fonction `occurrences` :
   - Invariant de la boucle externe : $\text{Inv}(i)$: $\forall k < i$ `occ.(k)` contient le nombre d'occurrences de `t.(k)` dans `t`.
   - Invariant de la boucle interne : $\text{Inv}'(j)$: `occ.(i)` contient le nombre d'occurrences de `t.(i)` parmi `t.(0)`, ..., `t.(j-1)`.

   fonction `indice_max` : $\text{Inv}''(i)$: $\forall k < i$, `t.(k)` $\leq$ `t.(!imax)`.

5. Une fois le tableau trié, on peut trouver la valeur majoritaire en ne parcourant qu'une seule fois chaque case du tableau, car les valeurs identiques apparaissent désormais consécutivement dans le tableau. Ainsi, bien que la fonction ci-dessous possède deux boucles `while` imbriquées, on ne passe qu'une seule fois par chaque case du tableau, et la complexité de la boucle while externe est donc linéaire. Au final, la solution proposée ci-dessous à une complexité en $O(n \log n)$, ce qui est mieux qu'à la question 1 (l'opération la plus coûteuse est de trier le tableau).

```
1  let valeur_majoritaire' t =
2  let t' = trier t in (* complexite en O(n log n) *)
3    let n = Array.length t' in
4    let i = ref 0 in (* les valeurs de t' entre les indices *)
5    let j = ref 1 in (* i et j seront toutes egales *)
6    let v = ref t'.(0) in (* contiendra la valeur majoritaire *)
7    let occ_max = ref 1 in (* contiendra le nombre d'occurrences de v *)
8    while !i < n do (* complexite en O(n) *)
9      let v_i = t'.(!i) in
10     while !j < n && v_i = t'.(!j) do
11       incr j (* tant que la valeur ne change pas : on avance *)
12     done ;
13     let occ_i = !j - !i in (* nombre d'occurrences de la valeur courante *)
14     if occ_i > !occ_max then
15       begin
16         occ_max := occ_i ;
17         v := v_i
18       end ;
19     (* on passe a la prochaine plage de t' *)
20     i := !j ;
21     j := !i + 1
22   done ;
23   !v
24 ;;
```

3 Recherche par dichotomie

Exercice 3 :

1. La fonction `dicho` prend en arguments un tableau `t` et un élément `x` comparable aux éléments de `t`, et renvoie un booléen. `val dicho : 'a array -> 'a -> bool = <fun>`
2. Trier le tableau avec la fonction précédente nécessiterait un temps en $O(n \log n)$. La fonction dicho est plus efficace : $O(\log n)$, et perdrait alors son intérêt.
3. Supposons que P soit vraie en haut du corps de la boucle, et supposons que `x` apparaisse dans `t`. Donc `x` apparait entre les indices g et d. On a $m = \lfloor \frac{(g+d)}{2} \rfloor$, et notons $v_m =$ `t.(m)`. Comme le tableau est trié :
   - soit $x \leq v_m$, et x se trouve forcément entre g et m ;
   - soit $x > v_m$, et x se trouve forcément entre m et d.

   Dans tous les cas, P est vraie en bas du corps de la boucle.
4. `dicho [|1;2|] 2 ;;` ne termine pas :
   - au départ, $g = 0$ et $d = 1$;
   - on a donc $m = 0$, et lorsqu'on passe dans la ligne 8, g reste à 0 ;
   - on a donc $g = 0$ et $d = 1$ indéfiniment, et la boucle ne s'arrête jamais.
5. Il suffit de modifier la ligne 8 par : `else g := m + 1`
6. **Terminaison.** Notons u_n la valeur de $d - g$ après n passages dans la boucle. (u_n) est une suite à valeurs dans $\mathbb{N}$ et tend vers 0 (car $u_{n+1} < u_n/2$, on peut donc montrer que $0 \leq u_n \leq \frac{u_0}{2^n}$). Donc à partir d'un certain rang, (u_n) est stationnaire, égale à 0. De plus, la boucle while termine lorsque $u_n = 0$. Donc notre fonction termine.
   Correction. La propriété P est vraie avant la boucle (car g et d sont alors les indices extrémaux du tableau). De plus, P est toujours préservée à chaque passage dans la boucle malgré notre modification : si $x > v_m$, alors x n'est pas non plus à l'indice m, donc il se trouve forcément entre les

indices $m+1$ et d. Ainsi, à la fin de l'algorithme, on a $g = d$, et P est toujours vérifiée, donc x apparait dans `t` ssi x apparait à l'indice g. Le booléen renvoyé est donc correct.

7. (a)

```
let tricho t x =
  let n = Array.length t in
  let rec aux t g d = match (d=g) with
    | true -> t.(g) = x
    | false when t.(g+(d-g)/3) > x -> aux t g (g+((d-g)/3)-1) (* gauche *)
    | false when t.(g+2*(d-g)/3) < x -> aux t (g+(2*(d-g)/3)+1) d (* droite *)
    | _ -> aux t (g+(d-g)/3) (g+2*(d-g)/3) (* milieu *)
  in aux t 0 (n-1)
;;
```

(b) Notons g_k et d_k les valeurs successives de `g` et `d` lors des appels récursifs de `aux`, et notons $l_k = d_k - g_k$. On a $l_0 = n-1$, et $l_{k+1} \leq l_k/3$ (peu importe le cas du pattern matching utilisé). Ainsi, on obtient facilement par récurrence que $\forall k, 0 \leq u_k \leq \frac{n-1}{3^k}$. Donc $u_k \to 0$, et il existe un rang k_t à partir duquel $u_{k_t} = 0$ (car (u_k) est à valeurs dans $\mathbb{N}$).
De plus, on a $u_{k_t} = 0$ dès que $\frac{n-1}{3^{k_t}} < 1 \iff n-1 < 3^{k_t} \iff \log_3(n-1) < k_t$.
Donc $k_t = \lceil \log_3(n-1) \rceil = O(\log n)$.
La fonction `tricho` a donc une complexité en $O(\log n)$.