OPTION INFORMATIQUE
Concours blanc
Corrigé

1 Pile avec oubli

On représente une pile de capacité c par un tableau de taille $c+2$. Les c premières cases contiennent les éléments de la pile, dans l'ordre mais avec possiblement de la circularité. La case d'indice c indique la taille de la pile, et la case d'indice $c+1$ indique la position du sommet.
Par exemple, la pile $(5,2,1,3)$ (où 5 est le sommet) de capacité 10 peut être représentée par le tableau `[|3;1;2;5;0;0;0;0;0;0;4;3|]`, ou également par le tableau `[|2;5;8;2;0;6;4;6;3;1;4;1|]`

```
let c = 30;;

let creer_pile () = Array.make (c+2) 0;;

let est_vide p =
    let c = Array.length p - 2 in p.(c) = 0;;

let empiler p v =
    let c = Array.length p - 2 in
    let i = p.(c+1) in
    let i' = (i + 1) mod c in
    p.(i') <- v;
    p.(c+1) <- i';
    if p.(c) < c then p.(c) <- p.(c) + 1;;

let depiler p =
    if est_vide p then failwith "depilement de pile vide" else
    let c = Array.length p - 2 in
    let i =p.(c+1) in
    let v = p.(i) in
    let i' = (i - 1) mod c in
    p.(c+1) <- i';
    p.(c) <- p.(c) - 1;
    v;;
```

2 Conversion d'une liste en matrice

```
let matrice l n =
    let m = Array.make_matrix n n 0 in
    let rec aux i j l =
        match i,j,l with
            _,_,[] -> ()
            |_ when i > n-1 -> ()
            |_ when j > n-1 -> aux (i+1) 0 l
            |i,j,a::q -> (m.(i).(j) <- a ; aux i (j+1) q)
    in
    aux 0 0 l; m;;
```

3 Arbres et arités

1.
```
let rec somme a =
    match a with
      V -> 0
    | N(x,l) -> x + somme_fratrie l
and somme_fratrie l  =
    match l with
      [] -> 0
    | a::q -> somme a + somme_fratrie q;;
```

2.
```
let rec arite_max a =
    match a with
      V -> 0
    | N(x,l) -> max (arite l) (arite_max_fratrie l)
and arite l = (* renvoie le nombre d'arbres non vides dans l *)
    match l with
      [] -> 0
    | V::q -> arite q
    | N(_)::q -> 1 + arite q
and arite_max_fratrie l = (* renvoie l'arité maximale des arbres dans l *)
    match l with
      [] -> 0
    | a' :: q -> max (arite_max a') (arite_max_fratrie q);;
```

3.
```
let rec est_entier_fratrie l n =
(* teste si tous les noeuds des arbres de l sont d'arité 0 ou n *)
    match l with
      [] -> true
    | V :: q -> est_entier_fratrie q n
    | N(x,l')::q -> let n' = arite l' in
                    (n' = n || n' = 0) && est_entier_fratrie l' n
                    && est_entier_fratrie q n;;
let est_entier a =
    match a with
      V -> true
    | N(x,l) -> let  n = arite l in
                est_entier_fratrie l n;;
```

4 Problème de l'arrêt

1. (a) `f1` termine sur les entiers positifs pairs.

   (b) pour $k \in \mathbb{N}$, on pose P_k : "`f1` termine sur l'entier $2k$". On montre $\forall k \in \mathbb{N}, P_k$ par récurrence sur k :
   - Initialisation : `f1` termine bien sur 0.
   - Hérédité : soit $k \in \mathbb{N}$ vérifiant P_k. L'appel `f1` $2(k+1)$ effectue l'appel récursif `f1` $2k$, qui termine par hypothèse de récurrence. P_{k+1} est donc vérifié.

   En conclusion, on a bien $\forall k \in \mathbb{N}, P_k$

2. `f2` termine sur tous les entiers inférieurs ou égaux à 10.

   Soit $n \leq 10$.

   On utilise pour prouver la terminaison de la boucle `while` le variant de boucle $V = 10 - i$. La propriété $i \leq 10$ est un invariant de boucle : elle est vraie au début et conservée à chaque itération. En conséquence, V est au début de chaque itération à valeur dans $\mathbb{N}$, et a strictement diminué à la fin de l'itération. V est donc bien un variant, ce qui garantit la terminaison.

3. `let rec infinie x = infinie x`
   Ironiquement, le fait d'être récursive *terminale* permet à cette fonction de ne terminer vraiment jamais, pas même par saturation de la pile d'appel.
4. (a) `arret : string -> 'a -> bool`.
   (b)
```
let bizarre code_f x =
    if arret code_f x then infinie x else 0;;
```
   (c) `let paradoxe code_f = bizarre code_f code_f;;`
   (d) Par définition, `paradoxe code_paradoxe` termine si et seulement si `paradoxe code_paradoxe` ne termine pas, ce qui est absurde. On en déduit que la fonction `arret` n'existe pas.

   Remarque : on pourrait avoir l'impression que le paradoxe a quelque chose à voir avec le fait que le code de `paradoxe` ne contienne pas toute l'information nécessaire pour analyser l'exécution, puisqu'il y a un appel à `bizarre`, qui elle-même appelle `arret`. Ce problème se résout facilement : on pourrait faire de `arret` et de `bizarre` des fonctions locales à `paradoxe`, de sorte que `code_paradoxe` soit auto-suffisant, et le paradoxe subsisterait. Cette démonstration s'adapte à tout langage Turing-complet, On dit que le problème de l'arrêt d'un programme est *indécidable*.

5 Recherche de valeur absolument majoritaire

5.1 Méthode naïve

1. (a)
```
let occurrences tab n =
    let r = ref 0 in
    for i = 0 to (Array.length tab - 1) do
        if tab.(i) = n then incr r;
        done;
    !r;;
```
   (b) il y a N itérations de la boucle `for`, chacune en $O(1)$, d'où une complexité en $O(N)$.
2. (a)
```
let elu1 k tab =
    let n = ref 0 in
    while !n<k && occurrences tab !n <= Array.length tab / 2 do
        incr n;
        done;
    if !n = k then -1 else !n;;
```
   (b) Dans le pire des cas, la boucle `while` est exécutée k fois, chaque itération étant en $O(N)$ (à cause du calcul de la condition d'entrée), d'où une complexité en $O(kN)$.

5.2 Par le tableau des occurrences

1.
```
let tableau_occ k tab =
    let occ = Array.make k 0 in
    for i = 0 to (Array.length tab - 1) do
        occ.(tab.(i)) <- occ.(tab.(i)) + 1;
        done;
    occ;;
```
2.
```
let max tab =
    let m = ref tab.(0) in
    let j = ref 0 in
    for i = 1 to (Array.length tab - 1) do
        if tab.(i) > !m then begin m := tab.(i); j:= i end;
        done;
    !m,!j;;
```

```
let elu2 k tab =
    let occ = tableau_occ k tab in
    let m,j = max occ in
    if m>(Array.length tab)/2 then j else -1;;
```

3. La complexité de `tableau_occ` est en $O(k)$ (création du tableau `occ`) $+O(N)$ (boucle `for`) $= O(k+N)$.
   La complexité de l'appel `max occ` est clairement en $O(k)$.
   La complexité de `elu2` est donc en $O(k+N)+O(k) = O(k+N)$.

5.3 *Diviser pour régner* simple

1. Par contraposée : Soit $n \in \mathbb{N}$ élue ni par $T[0:m]$ ni par $T[m:N]$. Cela signifie que n apparaît au plus $m/2$ fois dans $T[0:m]$ et au plus $(N-m)/2$ fois dans $T[m:N]$, donc au plus $m/2+(N-m)/2 = N/2$ fois dans T, et n'est donc pas élue par T.
2. ```
   let occurrences2 tab n i j =
       let r = ref 0 in
       for k = 0 to j - 1 do
           if tab.(i) = n then incr r;
           done;
       !r;;
   ```
3. (a) ```
   let rec dpr_simple tab i j =
       if j-i <= 1 then (tab.(i),1)
       else begin
               let m = (i+j)/2 in
               let a , ag = dpr_simple tab i m in
               let b , bd = dpr_simple tab m j in
               if (a,b) = (-1,-1) then (-1,0)
               else begin
                   let qa = ag + occurrences2 tab a m j in
                   if qa > (j-i)/2 then (a ,qa)
                   else begin
                       let qb = bd + occurrences2 tab b i m in
                       if qb > (j-i)/2 then (b , qb)
                           else (-1,0)
                       end
                   end
               end;;
   ```

   (b) Soit $N = j-i$. Si $N \leq 1$, `dpr_simple tab i j` termine clairement. On démontre la terminaison dans les autres cas par récurrence forte sur N :

   Initialisation :
   Si $N = 1$, déjà vu.

   Hérédité :
   Soit $N > 1$ tel que $\forall z \in [\![1; N-1]\!]$, $j-i = z \Rightarrow$ `dpr_simple tab i j` termine.

   Soient i, j tels que $j-i = N$. Soit $m = \lfloor (i+j)/2 \rfloor$.
   On a $(i+j)/2 - 1 < m \leq (i+j)/2$, d'où $N/2 - 1 < m-i \leq N/2$ et $m-i \in [\![1, N-1]\!]$

   Par hypothèse de récurrence, l'appel `dpr_simple tab i m` termine donc.

   De même, $N/2 \leq j-m < N/2+1$ d'où $j-m \in [\![1, N-1]\!]$ donc par hypothèse de récurrence, l'appel `dpr_simple tab m j` termine.

   On en déduit que `dpr_simple tab i j` termine, ce qui achève la récurrence.

4.
```
let elu3 tab =
    let a,q = dpr_simple tab 0 (Array.length tab - 1) in a;;
```

5. Soit $N = j - i$. La complexité de `dpr_simple` vérifie la relation de récurrence

   $C(N) = 2C(N/2)$ (coût des appels récursifs) $+ O(N)$ (coût des 2 appels à occurrences2)

   On en déduit que la complexité de `elu3` est elle-même en $O(N \log N)$.

5.4 *Diviser pour régner* avancé

1. 3 est un postulant de T pour 5 car $5 > 8/2$, 3 apparait au plus 5 fois dans T et les autres valeurs apparaissent au plus $8 - 5 = 3$ fois dans T.
2. Supposons a élu par T. Soit n le nombre d'occurrences de a dans T. Par définition, $n > N/2$, a apparaît au plus n fois dans T (en fait exactement n fois), et il y a $N-n$ cases non occupées par a dans T, donc toute autre valeur apparaît au plus $N - n$ fois dans T. a est donc un postulant de T pour n, donc a est un postulant de T.
3. Supposons que a est un postulant de T pour une valeur n. Alors $n > N/2$ et $N - n < N/2$ et toute autre valeur apparaît au plus $N - n$ fois dans T, donc ne peut pas être élue par T.
4. On a vu que `[|1;2;3;4;3;2;3;3|]` a un postulant alors qu'il n'a pas d'élu.

   Le tableau `[|0;1|]` n'a pas de postulant. Par exemple, si 0 était postulant, ce ne pourrait être que pour la valeur 2, mais 1 apparaît plus de 0 fois dans le tableau.

5. (a) Soit $n = l + \lceil m/2 \rceil$ (où $\lceil x \rceil$ dénote la partie entière supérieure de x).
   - On a $l > m/2$, donc $n > m/2 + \lceil m/2 \rceil \geq m$
   - a apparaît au plus l fois dans $T[0 : m]$ et au plus $\lfloor m/2 \rfloor$ fois dans $T[m : N]$ (car ce tableau n'élit pas a), donc a apparaît au plus n fois dans T.
   - Soit $b \neq a$. b apparaît au plus $m - l$ fois dans $T[0 : m]$ et au plus $\lfloor m/2 \rfloor$ fois dans $T[m : N]$. On a bien :

   $$m - l + \lfloor m/2 \rfloor \leq 3m/2 - l < 2m - l - \lceil m/2 \rceil = N - n$$

   On en déduit que a est postulant de T pour n.

   (b) i. Soit $n = l + q$.
   - $l < m/2$ et $q < m/2$ donc $n = l + q < m$.
   - a apparaît bien au plus $l + q$ fois dans T.
   - Soit $c \neq a$. c apparaît au plus $m - l$ fois dans $T[0 : m]$ et au plus $m - q$ fois dans $T[m : N]$ donc au plus $m - l + m - q = N - n$ fois dans T.

   On en déduit que a est postulant de T pour n.

   ii.
   - On a bien $n = m + q - l > m$.
   - b apparaît au plus q fois dans $T[m : N]$ et au plus $m - l$ fois dans $T[0 : m]$, donc au plus n fois dans T.
   - — a apparaît au plus l fois dans $T[0 : m]$ et au plus $m - q$ fois dans $T[m : N]$, donc au plus $l + m - q = N - n$ fois dans T.
     — Soit $c \notin \{a, b\}$. c apparaît au plus $m - l$ fois dans $T[0 : m]$ et au plus $m - q$ fois dans $T[m : N]$, donc au plus $m - l + m - q \leq m - l + m - q + 2l - m = m + l - q = N - n$ fois dans T.

   On en déduit que b est postulant de T pour n.

   iii. Soit $c \in \mathbb{N}$.
   - Si $c \notin \{a, b\}$, c n'est pas élu dans $T[0 : m]$ (car a est un postulant) et c n'est pas élu dans $T[m : N]$ (car b est un postulant), donc c n'est pas élu dans T d'après la question 1 de la partie précédente.
   - Si $c = a$. c apparaît au plus l fois dans $T[0 : m]$ et au plus $m - q = m - l$ fois dans $T[m : N]$, donc c apparaît au plus m fois dans T et n'est donc pas élu par T.
   - Similairement, si $c = b$, c n'est pas élu par T.

En conclusion, T n'admet pas d'élu.

6.
```
let rec postulant tab i j =
      let n = j-i in
      if n <= 1 then (tab.(i),1)
      else begin
              let m = n/2 in
              let m2 = m+i in
              let (a,l) = postulant tab i m2 in
              let (b,q) = postulant tab m2 j in
              match (a,b) with
                  (-1,-1) -> (-1,0)
                  |(-1,_) -> (b,q+m/2)
                  |(_,-1) -> (a,l+m/2)
                  |_ when a=b -> (a,l+q)
                  |_ when q>l -> (b,m+q-l)
                  |_ -> (a,m+l-q)
         end;;
```

7.
```
let elu4 tab =
      let n = Array.length tab in
      let a,l = postulant tab 0 (n - 1) in
      if occurrences tab a > n/2 then a else -1;;
```

8. La complexité de la fonction `postulant` vérifie la relation $C(N) = 2C(N/2) + O(1)$, où $N = j - i$.
   On en déduit $C(N) = O(N)$.
   `elu4` a la même complexité, l'appel à `occurrences` étant lui-même en $O(N)$.