OPTION INFORMATIQUE
Devoir surveillé 2
Corrigé

1 Algorithmes de tri

1.1 Test de tri

1. ```
   let rec liste_triee l =
       match l with
         [] -> true
       | [a] -> true
       |a::b::q -> a<=b && liste_triee (b::q);;
   ```

2. ```
   let tableau_trie t =
       let n = Array.length t in
       let i = ref 0 in
       while !i < (n-1) && t.(!i) <= t.(!i+1) do incr i done;
       !i = n-1;;
   ```

3. Dans le pire cas, atteint quand la liste est triée, la complexité de la fonction `liste_triee` vérifie la relation de récurrence $C(n) = C(n-1) + O(1)$, d'où $C(n) = O(n)$, où n est la longueur de la liste. Dans le meilleur cas, atteint quand le premier élément est strictement supérieur au second, on a directement $C(n) = O(1)$.

   Dans le pire cas, atteint quand le tableau est trié, on fait $n-1$ passages dans la boucle `while`, chaque passage en $O(1)$, d'où une complexité de `tableau_trie` en $O(n)$, où n est la longueur du tableau. Dans le meilleur cas, atteint quand le premier élément est strictement supérieur au second, on ne fait aucun passage dans le `while` et la complexité est en $O(1)$.

1.2 Tri par insertion

1. ```
   let rec insertion l x =
       match l with
         []-> [x]
       | a::q when x<a -> x::l
       | a::q -> a::(insertion q x);;
   ```

2. ```
   let rec tri_insertion_liste l =
       match l with
         []-> []
       | x::q -> insertion (tri_insertion_liste q) x;;
   ```

3. Dans le meilleur cas, atteint quand x est inférieur à tous les éléments de l, `insertion x l` a une complexité en $O(1)$. Dans le pire cas, atteint quand x est supérieur à tous les éléments de l, la complexité de l'insertion est en $O(n)$ où n est la longueur de la liste.

   Dans le meilleur cas, la relation de récurrence de la complexité de `tri_insertion` est donc $C(n) = C(n-1) + O(1)$, d'où $C(n) = O(n)$. Ce meilleur cas est atteint pour des listes initialement déjà triées.

   Dans le pire cas, la relation de récurrence de la complexité de `tri_insertion` est donc $C(n) = C(n-1) + O(n)$, d'où $C(n) = O(n^2)$. Ce pire cas est atteint pour des listes initialement rangées dans l'ordre décroissant.

4. ```
   let tri_insertion_tableau t =
       let n = Array.length t in
       for i = 1 to (n-1) do
   ```
   ```

```
        let j = ref i in
        let x = t.(i) in
        while !j>0 && t.(!j -1) > x do
            t.(!j) <- t.(!j - 1);
            decr j
        done;
        t.(!j) <- x
    done;;
```

5. Dans le meilleur cas, atteint quand le tableau est trié, on ne fait aucun passage dans la boucle `while`, d'où une complexité en $O(n)$, où n est la longueur du tableau.
   Dans le pire cas, atteint quand les éléments du tableau sont dans un ordre strictement décroissant, on fait i passages dans la boucle `while` à chaque passage de la boucle `for`, soit $\frac{n(n-1)}{2}$ passages au total, d'où une complexité en $O(n^2)$.
6. On pose I1 : les éléments de t entre les indices 0 et $i-1$ sont triés. On montre que I1 est un invariant de la boucle `for`.
   - Initialement, i vaut 1, I1 ne porte que sur un seul élément, donc est vrai
   - Supposons I1 vrai au début d'une itération, et montrons qu'il est encore vrai en fin d'itération. Pour analyser la boucle `while`, on pose alors I2 : t est trié des indices 0 à $j-1$, et des indices $j+1$ à i, et si $j+1 \leq i$, alors $x \leq$ `t.(j+1)`. On montre que I2 est un invariant de la boucle `while` :
     — Initialement, j vaut i, et t est bien trié entre les indices 0 à $i-1$ d'après I1.
     — Supposons I2 vrai au début d'une itération, et montrons qu'il est encore vrai à la fin de cette itération.
     En notant j' la valeur de cette variable en fin d'itération, on a $j' = j-1$. t est bien trié des indices 0 à $j'-1$ et de $j'+1$ à i (ce sont les mêmes éléments que ceux décrits par l'invariant en début d'itération). De plus on a maintenant $j'+1 \leq i$, et, d'après la condition de boucle, `x < t.(j-1)` donc `x < t.(j)` (d'après l'instruction `t.(j) <- t.(j-1)`), donc `x < t.(j' + 1)`.
     En conclusion, I2 est un invariant de boucle du `while`. En sortie de boucle (garantie car j est un variant de boucle), on a $j = 0$ ou `t.(j-1) <= x`. Dans les deux cas, après l'écriture `t.(j) <- x`, on a bien t trié des indices 0 à i, ie I1 est encore vrai en fin d'itération.
   - Finalement, I1 est encore vrai en sortie de la boucle `for`, avec $i = n$. La fonction est donc bien correcte.

1.3 Tri fusion

1. ```
   let rec diviser l =
       match l with
         []-> [],[]
       | [a] -> [a],[]
       | a::b::q -> let q1,q2 = diviser q in a::q1,b::q2;;
   ```

2. ```
   let rec fusion l1 l2 =
       match l1,l2 with
         [],_-> l2
       | _,[] -> l1
       | a::q1 , b::q2 -> if a<b then a::(fusion q1 l2) else b::(fusion l1 q2);;
   ```

   La complexité de la fonction fusion vérifie la relation de récurrence $C(n) = C(n-1) + O(1)$, et est donc bien en $O(n)$.

3. ```
   let rec tri_fusion l =
       match l with
   ```
   ```

```
  [] -> []
| [a] -> [a]
| _ -> let l1,l2 = diviser l in
       fusion (tri_fusion l1) (tri_fusion l2);;
```

4. En notant $n = len(l)$ et $n' = n/2$, la complexité de `diviser` vérifie la relation $C(n') = C(n'-1) + O(1)$, et est donc en $O(n')$, ie en $O(n)$.

   La complexité de `tri_fusion` vérifie la relation de récurrence $C(n) = 2C(n/2) + O(n)$. D'après le théorème maître, la complexité est donc en $O(n \log n)$.

1.4 Tri rapide sur liste

1. ```
   let rec repartition l x =
       match l with
         [] -> [],[]
       | a::q -> let q1,q2 = repartition q x in
                 if a<x then a::q1,q2 else q1,a::q2;;
   ```

2. ```
   let rec concat l1 l2 =
       match l1 with
         [] -> l2
       | a::q -> a::(concat q l2);;
   ```

3. ```
   let rec tri_rapide_liste l =
       match l with
         [] -> []
       | a::q -> let l1,l2 = repartition q a in
                 concat (tri_rapide_liste l1) (a::(tri_rapide_liste l2));;
   ```

4. La complexité de `repartition` vérifie la relation $C(n) = C(n-1)+O(1)$, d'où $C(n) = O(n)$, où $n = len(l)$.

   Similairement, la complexité de `concat` vérifie la relation $C(n) = C(n-1) + O(1)$, d'où $C(n) = O(n)$, où $n = len(l1)$.

5. Dans ce cas, la complexité vérifie la relation $C(n) = 2C(n/2) + O(n)$, et est donc en $O(n \log n)$.

6. Dans ce cas, sur les deux appels récursifs, l'un se fera sur une liste vide et sera donc en $O(1)$, et l'autre se fera sur une liste contenant tous les éléments sauf le premier. la complexité vérifie alors la relation $C(n) = C(n-1) + O(n)$, et est donc en $O(n^2)$.

   Ce cas est par exemple atteint lorsque la liste initiale est déjà triée.

7. On montre les deux propriétés en même temps, par récurrence sur n :

   Pour $n \in \mathbb{N}$, on pose $P(n)$: si l est une liste de longueur n, alors `tri_rapide_liste(l)` termine et renvoie une liste triée formée des mêmes éléments que l.

   On montre $\forall n \in \mathbb{N}, P(n)$ par récurrence forte sur n :

   - Initialisation : si $n = 0$, la propriété est vraie par lecture immédiate du programme.
   - Hérédité : Soit $n > 0$ tel que $\forall 0 \leq k < n, P(k)$ est vraie. Montrons $P(n)$. Soit `l` une liste de longueur n, de la forme `a::q`. L'appel `repartition q a` termine et renvoie deux listes `l1` et `l2` contenant les éléments de `q` et telles que les éléments de `l1` sont strictement inférieurs à `a`, qui est inférieur ou égal aux éléments de `l2`. Par hypothèse de récurrence, `tri_rapide_liste l1` et `tri_rapide_liste l2` sont calculées en temps fini, vérifient les mêmes propriétés et sont triées. Enfin, l'appel à concaténation termine et forme bien une liste triée contenant les éléments de l. En conclusion, l'appel initial termine et est correct.
   ```

1.5 Tri rapide en place

1. ```
   let permuter t i j =
       let a = t.(i) in
       t.(i) <- t.(j);
       t.(j) <- a;;

   let repartition_en_place t g d =
       let p = ref g in
       for i = g to (d-1) do
           if t.(i) <= t.(d)
           then begin
               permuter t !p i;
               incr p
               end
       done;
       permuter t !p d;
       !p;;
   ```

   Il y a bien $d-g$ passages dans le `for`, chaque passage en $O(1)$, d'où la complexité demandée.

2. ```
   let  tri_rapide_en_place t =
       let n = Array.length t in
       let rec aux a b =
           if b-a > 0 then begin
               let p = repartition_en_place t a b in
               aux a (p-1);
               aux (p+1) b
               end
       in
       aux 0 (n-1);;
   ```

1.6 Tri par comptage

1. ```
   let tri_comptage t =
       let n = Array.length t in
       if n <> 0 then begin
       let m1 = ref t.(0) in
       let m2 = ref t.(0) in
       for i = 1 to (n-1) do
           if t.(i) < !m1 then m1 := t.(i);
           if t.(i) > !m2 then m2 := t.(i);
       done;
       let occ = Array.make (!m2 - !m1+1) 0 in
       for i = 0 to (n-1) do
           occ.(t.(i) - !m1) <- occ.(t.(i) - !m1) + 1
       done;
       let k = ref 0 in
       for i = !m1 to !m2 do
           for j = 1 to occ.(i - !m1) do
               t.(!k) <- i;
               incr k
           done
       done
       end;;
   ```

2. Les deux première boucle `for` sont en $O(n)$. La création de `occ` est en $O(M-m+1)$. La somme cumulée des passages dans la boucle `for j = 1 to occ.(i - !m1)` vaut `n` par
   ```

définition de `occ`, donc les boucles `for` imbriquées sont en $O(n+e)$, d'où une complexité totale en $O(n+e)$.

3. Ce tri utilise une quantité de mémoire en $O(e)$. Il n'est donc pas en place.
4. Dans la suite, on note à la python `t[i]` pour la partie de `t` entre les indices 0 et $i-1$.
   La première boucle a pour invariant : m1 est inférieur ou égal, et m2 supérieur ou égal, à chaque élément de `t[i]`.
   La seconde boucle a pour invariant : $\forall k \in [\![m1, m2]\!]$, `occ.(k)` contient le nombre d'occurrences de k dans `t[i]`.
   La troisième boucle `for` (verb—for i = !m1 to !m2—) a pour invariant : `t[k]` est trié et contient toutes les valeurs entre m1 et $i-1$ présentes initialement dans `t`.
   La quatrième boucle `for` (verb—for j = 1 to occ.(i - !m1)—) a pour invariant : `t[s:k]` ne contient que la valeur i, où s est la somme du nombre d'occurrence des valeurs entre m1 et $i-1$.

2 Arbres et arités

1. ```
   let l1 = N(1, [N(2, [N(3, [])])]);;
   let l2 = N(1, [N(2, []); N(3, [])] );;
   ```
   `l1` a pour parcours postfixe `[3; 2; 1]`, et `l2` a pour parcours postfixe `[2; 3; 1]`
2. ```
   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;;
   ```
3. ```
   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);;
   ```
4. ```
   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;;
   ```

3 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*.

4 Produit matriciel

4.1 Algorithme naïf

1.
```
let produit a b =
    let na = Array.length a in
    let ma = Array.length a.(0) in
    let mb = Array.length b.(0) in
    let c = Array.make_matrix na mb 0. in
    for i = 0 to (na - 1) do
        for j = 0 to (mb - 1) do
            for k = 0 to (ma - 1) do
                c.(i).(j) <- c.(i).(j) +. a.(i).(k) *. b.(k).(j)
            done
        done
    done;
    c;;
```

2. Par lecture immédiate du programme, la complexité est en $O(nambma)$, où les matrices sont respectivement de dimensions $na \times ma$ et $ma \times mb$.

4.2 Diviser pour régner naïf

1. ```
   let decoupage a =
       let n = Array.length a in
       let m = n/2 in
       let a00 = Array.make_matrix m m 0. in
       let a01 = Array.make_matrix m m 0. in
       let a10 = Array.make_matrix m m 0. in
       let a11 = Array.make_matrix m m 0. in
       for i = 0 to m-1 do
           for j = 0 to m-1 do
               a00.(i).(j) <- a.(i).(j);
               a01.(i).(j) <- a.(i).(m+j);
               a10.(i).(j) <- a.(m+i).(j);
               a11.(i).(j) <- a.(m+i).(m+j)
           done
       done;
       a00, a01, a10, a11;;
   ```

2. ```
   let recollage a00 a01 a10 a11 =
       let m = Array.length a00 in
       let n = m*2 in
       let a = Array.make_matrix n n 0. in
       for i = 0 to m-1 do
           for j = 0 to m-1 do
               a.(i).(j) <- a00.(i).(j);
               a.(i).(m+j) <- a01.(i).(j);
               a.(m+i).(j) <- a10.(i).(j);
               a.(m+i).(m+j) <- a11.(i).(j)
           done
       done;
       a;;
   ```

3. ```
   let somme a b =
       let n = Array.length a in
       let m = Array.length a.(0) in
       let c = Array.make_matrix n m 0. in
       for i = 0 to n-1 do
           for j = 0 to m-1 do c.(i).(j) <- a.(i).(j) +. b.(i).(j)
           done
       done;
       c;;
   ```

4. Les fonctions `decoupage` et `recollage` sont en $O(n^2)$. La fonction `somme` est en $O(nm)$, où la dimension des matrices sommées est $n \times m$.

5. ```
   let rec produit_dpr a b =
       let n = Array.length a in
       if n=1 then [| [| a.(0).(0) *. b.(0).(0) |] |]
       else begin
           let a00, a01, a10, a11 = decoupage a in
           let b00, b01, b10, b11 = decoupage b in
           let c00 = somme (produit_dpr a00 b00) (produit_dpr a01 b10) in
           let c01 = somme (produit_dpr a00 b01) (produit_dpr a01 b11) in
           let c10 = somme (produit_dpr a10 b00) (produit_dpr a11 b10) in
           let c11 = somme (produit_dpr a10 b01) (produit_dpr a11 b11) in
           recollage c00 c01 c10 c11
       end;;
   ```

6. La complexité de `produit_dpr` vérifie la relation de récurrence $C(n) = 8C(n/2) + O(n^2)$. Le théorème maître appliqué pour $a = 8, b = 2, c = 2$ donne alors une complexité en $O(n^3)$, qui est la même que celle de l'algorithme naïf sur une matrice carrée. Cette application de la méthode diviser pour régner n'a donc pas permis d'améliorer la complexité.

4.3 Algorithme de Strassen

1. On trouve :
   - $C_{0,1} = M_3 + M_5$
   - $C_{1,0} = M_2 + M_4$
   - $C_{1,1} = M_1 - M_2 + M_3 + M_6$
2. Il nous faut une fonction `difference` similaire à la fonction `somme` :

```
let difference a b =
    let n = Array.length a in
    let m = Array.length a.(0) in
    let c = Array.make_matrix n m 0. in
    for i = 0 to n-1 do
        for j = 0 to m-1 do c.(i).(j) <- a.(i).(j) -. b.(i).(j)
        done
    done;
    c;;

let rec strassen a b =
    let n = Array.length a in
    if n=1 then [| [| a.(0).(0) *. b.(0).(0) |] |]
    else begin
        let a00, a01, a10, a11 = decoupage a in
        let b00, b01, b10, b11 = decoupage b in

        let m1 = strassen (somme a00 a11) (somme b00 b11) in
        let m2 = strassen (somme a10 a11) b00 in
        let m3 = strassen a00 (difference b01 b11) in
        let m4 = strassen a11 (difference b10 b00) in
        let m5 = strassen (somme a00 a01) b11 in
        let m6 = strassen (difference a10 a00) (somme b00 b01) in
        let m7 = strassen (difference a01 a11) (somme b10 b11) in

        let c00 = somme (difference (somme m1 m4) m5) m7 in
        let c01 = somme m3 m5 in
        let c10 = somme m2 m4 in
        let c11 = somme (somme (difference m1 m2) m3) m6 in

        recollage c00 c01 c10 c11
    end;;
```

3. On a la relation de récurrence $C(n) = 7C(n/2)+O(n^2)$, d'où $C(n) = O(n^{\log_2(7)})$. C'est bien une complexité strictement meilleure que celles des algorithmes précédents.