E3A 2019 MP option informatique

Proposition de corrigé par Quentin Fortier

Exercice 1

1. La recherche par dichotomie suppose la liste `l` triée et sert à savoir si un élément `e` y appartient. Pour cela, on compare `e` avec l'élément `m` au milieu de la liste: si `e < m` il faut chercher `e` dans la moitié gauche de `l`, sinon dans la partie droite. On réitère ce processus jusqu'à trouver `e` ou ne plus avoir d'élément à considérer.
   L'intérêt est que la complexité est alors logarithmique en la taille de `l` (au lieu de linéaire pour la méthode «naïve»).

2. C'est possible si on peut comparer des couples ou des chaînes de caractères. Avec Python (et OCaml) on peut comparer (dans l'ordre lexicographique) directement des couples ou des chaînes de caractères avec `<`.

3. Cela pourrait augmenter la complexité: passer de $O(\log_2(n))$ à $O(n\log_2(n))$ si le tri est en $O(n\log_2(n))$, par exemple. Notons qu'un tri peut être en $O(n\log_2(n))$ même sur une liste déjà triée.

4. Considérons une itération du `while`:

   - Si `liste[m] >= x`: si $\mathcal{P}$ était vrai, comme `liste` est triée, `x` apparaît avant l'indice `m`. Donc `x` apparaît dans `L[g:m+1]` et $\mathcal{P}$ reste vrai en remplaçant `d` par `m` (à l'itération suivante).
     Si $\mathcal{P}$ était faux alors `x` n'apparaît pas dans `L[g:d+1]` donc n'apparaît pas non plus dans `L[g:m+1]`.
   - Raisonnement similaire si `liste[m] < x`.

5. `dicho([0, 1], 1)` ne termine pas ($g=0, d=1, m=0$ et remplacer g par m ne change rien).

6. Remplacer `g = m` ligne (13) par `g = m + 1` (si `liste[m] < x`, on peut exclure `m` des indices à regarder).

7. Montrons alors que $d-g$ décroît strictement à chaque itération du `while`.
   Considérons une itération du `while`:

   - Si `liste[m] >= x`: on remplace d par m et $m-g = \lfloor\frac{g+d}{2}\rfloor - g \leq \frac{g+d}{2} - g = \frac{g+d-2g}{2} = \frac{d-g}{2} < d-g$ car $d-g>0$ (condition du `while`).
   - Si `liste[m] < x`: on remplace g par $m+1$ et $d-(m+1) = d-(\lfloor\frac{g+d}{2}\rfloor+1) < d-\frac{g+d}{2}$ (car $\lfloor\frac{g+d}{2}\rfloor > \frac{g+d}{2}-1$).
     D'où $d-(m+1) < \frac{d-g}{2} < d-g$.

   $d-g$ est un entier diminuant strictement à chaque itération du `while` donc devient négatif à un certain moment et la boucle s'arrête: dicho termine.
   Le prédicat/invariant de boucle de la question 4 est toujours vrai et montre que dicho est correct.

8. On découpe l'intervalle $[\![i,j]\!]$ en $[\![i,m_1]\!] \cup [\![m_1,m_2]\!] \cup [\![m_2,j]\!]$ où $m_1 = \lfloor i + \frac{j-i}{3}\rfloor = \lfloor\frac{2i+j}{3}\rfloor$ et $m_2 = \lfloor i + 2\times\frac{j-i}{3}\rfloor = \lfloor\frac{i+2j}{3}\rfloor$:

```python
def tricho(liste, x):
    def dans(i, j): # détermine si x est dans liste[i:j+1]
        if i >= j: return liste[i] == x
        m1, m2 = (2*i+j)//3, (i+2*j)//3
        if x <= liste[m1]:
            return dans(i, m1)
        if x <= liste[m2]:
            return dans(m1+1, m2)
        return dans(m2+1, j)
    return dans(0, len(liste)-1)
```

   Il est préférable d'utiliser des indices en arguments qui sont modifiés à chaque appel récursif plutôt que la liste elle-même car l'extraction de sous-liste avec `liste`$[i{:}j]$ est en complexité $O(j-i)$.

9. Soit $C(n)$ la complexité de `tricho(liste, x)` si `liste` est de taille n. Soit K le nombre maximum d'opérations réalisées lors d'un appel à `tricho(liste, x)` (sans compter les appels récursifs). Comme un appel récursif s'effectue sur une liste de taille divisée par (au moins) 3: $C(n) \overset{(*)}{\leq} K + C(\frac{n}{3}) \leq K + K + C(\frac{n}{9}) \leq ... \leq K \times p + C(\frac{n}{3^p})$, en appliquant p fois $(*)$.
Avec $p = \lceil \log_3(n) \rceil$, on trouve $C(n) = O(\log_3(n)) = \boxed{O(\log_2(n))}$ (tous les logarithmes sont égaux à une constante près car $\log_b(a) = \frac{\ln(a)}{\ln(b)}$).
La complexité en terme de O(...) est donc inchangée.
Remarque: si l'on s'intéresse au nombre exact d'opérations ce n'est pas clair: $\log_3(n) < \log_2(n)$ mais la constante cachée dans le O(...) augmente aussi (on fait plus de d'opérations élémentaires à chaque itération).

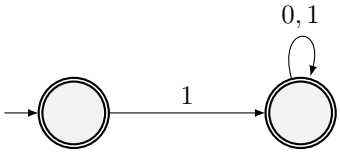
10. On peut effectuer une première recherche par dichotomie pour trouver la bonne colonne, puis une autre recherche par dichotomie sur cette colonne pour trouver la bonne ligne:

```python
def dicho_matrice(mat, x):
    n, p = len(mat), len(mat[0])
    # On localise la colonne en O(lg(p))
    g, d = 0, p-1
    while d-g > 0:
        m = (g+d)//2
        if mat[n-1][m] >= x:
            d = m
        else:
            g = m+1
    j = g
    # On localise la ligne en O(lg(n))
    g, d = 0, n-1
    while d-g > 0:
        m = (g+d)//2
        if mat[m][j] >= x:
            d = m
        else:
            g = m+1
    i = g
    if mat[i][j] == x:
        return (i,j)
    else:
        return (-1,-1)
```

La complexité est $O(\log(p))$ ($1^{\text{ère}}$ recherche dichotomique) + $O(\log(n))$ ($2^{\text{ème}}$ recherche dichotomique) = $\boxed{O(\log(\max(n,p)))}$.

Exercice 2

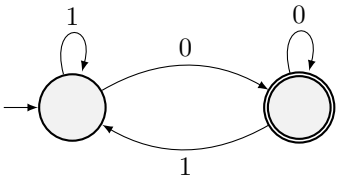
11. (a) 41 s'écrit $\boxed{101001}$ en base 2 (en utilisant par exemple la méthode par divisions successives par 2)
    (b) 10101010 représente l'entier $2^7 + 2^5 + 2^3 + 2 = 2 \times (64 + 16 + 4 + 1) = \boxed{170}$.
    (c) Un automate est local si deux transitions étiquetées par la même lettre aboutissent au même état.
    Un automate est standard s'il n'a qu'un seul état initial et qu'aucune transition n'aboutit sur cet état initial.
    (d)



    (e)

```
let langage_1 m = match m with
  | [] -> true
  | a::_ -> a;;
```

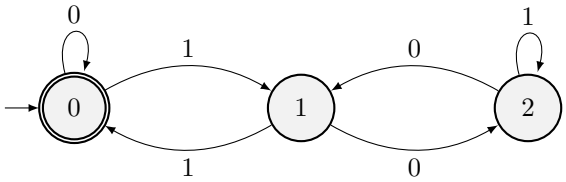
12. (a) Soit $P(L_2), S(L_2)$ les premières, dernières lettres des mots de L_2 et $F(L_2)$ les facteurs de longueur 2 des mots de L_2. On trouve que $P(L_2) = \{0,1\}$, $S(L_2) = \{0\}$, $F(L_2) = \{00, 01, 10, 11\}$.
    Alors, en notant $N(L) = A^2 \backslash F(L) = \emptyset$, $(P(L)A^* \cap A^*S(L)) \setminus (A^*N(L)A^*) = (A^* \cap A^*S(L)) \backslash \emptyset = A^*S(L) = L((0+1)^*0) = L_2$, ce qui montre que L_2 est local.
    (b)



(c)

```
let rec langage_2 m = match m with
  | [] -> false
  | [a] -> not a
  | a::u -> langage_2 u;;
```

13. (a)



(b)

```
let langage_3 m =
  let rec delta_etoile etat mot = match mot with
    | [] -> etat = 0
    | a::u -> delta_etoile ((2*etat + (if a then 1 else 0)) mod 3) u in
  delta_etoile 0 m;;
```

(c) $\mathcal{P}(1)$ signifie « $\forall w_1 \in A,\ \delta^*(0, w_1) = w_1 \mod 3$ » ce qui est vrai car $\delta^*(0, w_1) = \delta(\delta^*(0, \varepsilon), w_1) = \delta(0, w_1) = (2 \times 0 + w_1) \mod 3 = w_1 \mod 3$.
Soit $n \in \mathbb{N}^*$. Supposons $\mathcal{P}(n)$ et montrons $\mathcal{P}(n+1)$.
Soient $w_1, w_2, ..., w_{n+1} \in A$. Alors:

$$\delta^*(0, w_1 w_2 ... w_{n+1}) = \delta(\delta^*(0, w_1 w_2 ... w_n), w_{n+1}) \overset{\mathcal{P}(n)}{=} \delta(\sum_{k=1}^{n} w_k 2^{n-k}, w_{n+1})$$

$$= (2\sum_{k=1}^{n} w_k 2^{n-k} + w_{n+1}) \mod 3 = \sum_{k=1}^{n+1} w_k 2^{n+1-k} \mod 3$$

On a donc bien montré $\mathcal{P}(n+1)$. D'après le principe de récurrence, $\mathcal{P}(n)$ est donc vraie pour tout $n \in \mathbb{N}^*$.

14. (a) Clairement L_2 est l'ensemble des écritures en base 2 des entiers pairs. D'après 13c), L_3 est l'ensemble des écritures en base 2 des entiers congrus à 0 modulo 3 (c'est à dire multiples de 3).
D'après le théorème chinois, comme 2 et 3 sont premiers entre eux:

$$(k \equiv 0 \mod 2) \wedge (k \equiv 0 \mod 3) \iff k \equiv 0 \mod 6$$

Donc $L_1 \cap L_2 \cap L_3$ est l'ensemble des écritures en base 2 des entiers multiples de 6.

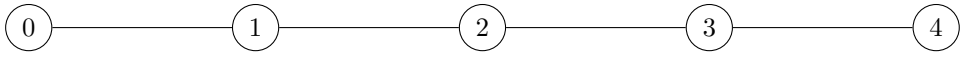
(b) D'après le cours, l'intersection de deux langages reconnaissables est reconnaissable. L_1, L_2, L_3 étant tous les trois reconnaissables (on a donné des automates pour chacun d'entre eux), $L_1 \cap L_2 \cap L_3$ est reconnaissable.
On peut aussi définir explicitement un automate reconnaissant $L_1 \cap L_2 \cap L_3$ par:

- l'ensemble d'états $Q = \{q_i, 0, 1, ..., 5\}$
- l'état initial q_i
- l'ensemble d'états finals $F = \{q_i, 0\}$
- la fonction de transition $\delta : Q \times A \longrightarrow Q$ définie par $\delta(q_i, 1) = 1$ (pour que le mot commence par 1) et, si $q \in \{0, 1, ..., 5\}$, $\delta(q, a) = (2q + a) \mod 6$

Exercice 3

15. G_1 a pour diamètre 3 et pour chemins maximaux $0, 2, 3, 4$; $0, 2, 3, 5$; $1, 2, 3, 4$; $1, 2, 3, 5$.
G_2 a pour diamètre 3 et pour chemins maximaux $0, 1, 2, 3$ ainsi que tous les chemins « symétriques » (il y en a beaucoup...).

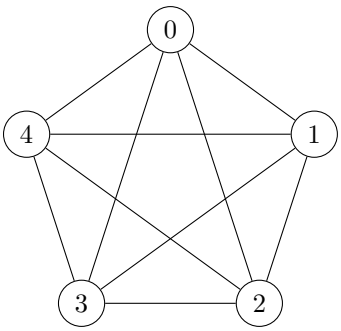
16. (a)



(b)

```
let diam_max n =
  let g = Array.make n [] in
  for i = 1 to n - 2 do
    g.(i) <- [i-1; i+1]
  done;
  if n > 1 then (g.(0) <- [1]; g.(n-1) <- [n-2]);
  g;;
```

17. (a) Un graphe complet (où toutes les arêtes possibles sont présentes) donne un diamètre de 1, qui est bien minimum:



(b)

```
let diam_min n =
  let g = Array.make n [] in
  for i = 0 to n - 1 do
    for j = 0 to n - 1 do
      if i <> j then g.(i) <- j::g.(i)
    done
  done;
  g;;
```

18. **Entrée**: un graphe G (orienté ou non) pondéré dont tous les poids sont positifs et un sommet s de G.
Sortie: la distance pondérée de s à chaque sommet de G (par exemple sous forme d'un tableau).
Étant donné un graphe non pondéré, on peut mettre un poids de 1 sur chaque arête et la distance pondérée est alors égale à la distance définie par l'énoncé. Il suffit ensuite d'appliquer une fois l'algorithme de Dijkstra depuis chaque sommet du graphe: la distance maximum trouvée est alors le diamètre.

19. On peut utiliser un parcours en largeur depuis chaque sommet du graphe, ce qui permet aussi d'obtenir toutes les distances donc le diamètre.

20. On sait d'après le cours que la complexité de l'algorithme de Dijkstra est plus élevée que celle du parcours en largeur. Comme dans les deux cas on applique l'algorithme autant de fois qu'il y a de sommets, il est préférable d'utiliser des parcours en largeur.
Plus précisément, si G est un graphe représenté par liste d'adjacence avec n sommets et m arêtes alors un parcours en largeur est en $O(n+m)$ donc la méthode de la question 19 est en $O(n(n+m))$ alors qu'une application de l'algorithme de Dijkstra est en $O(m\log(n))$ – si l'on utilise une file de priorité avec ajout et mise à jour en complexité logarithmique – ce qui donne une complexité totale $O(mn\log(n))$.

21. `Noeud (0, Noeud (1, Noeud (2, Noeud (4, Feuille, Feuille), Feuille),`
`Noeud (3, Noeud (5, Feuille, Feuille), Noeud (6, Feuille, Feuille))),`
`Feuille)`
Le diamètre de $G_{\mathcal{A}}$ est 4 et ses chemins maximaux sont $4,2,1,3,6$ et $4,2,1,3,5$

22. Soit $\mathcal{A}$ un arbre binaire enraciné en s et dont l'ensemble des noeuds est N. La fonction qui à chaque noeud v de $N\backslash\{s\}$ associe l'arc aboutissant en v est une bijection de $N\backslash\{s\}$ vers l'ensemble des arcs de $\mathcal{A}$.
On en déduit donc que $\boxed{r = n-1}$.
Remarque: on peut aussi démontrer ce résultat par récurrence (sur le nombre de noeuds, par exemple).

23.

```
let rec nb_noeuds a = match a with
  | Feuille -> 0
  | Noeud(r, g, d) -> 1 + nb_noeuds g + nb_noeuds d;;
```

24.

```
let numerotation a =
  let compteur = ref (-1) in
  let rec aux = function
    | Feuille -> Feuille
    | Noeud(_, g, d) -> (incr compteur;
                         let r = !compteur in
                         Noeud(r, aux g, aux d)) in
  aux a;;
```

Remarque technique: écrire `Noeud(!compteur, aux g, aux d)` ci-dessus ne fonctionnerait pas car les arguments sont évalués de droite à gauche (essayer `(print_int 1, print_int 2);;` par exemple, même si cela peut dépendre de la version de Caml). Ainsi `aux d` et `aux g` seraient d'abord évalués (ce qui modifie `compteur`) donc la valeur de `!compteur` ne serait plus la bonne.

25. On parcourt l'arbre en reliant chaque noeud avec son père:

```
let arbre_vers_graphe a =
  let ga = Array.make (nb_noeuds a) [] in
  let rec aux p = function (* p est le père de r *)
    | Feuille -> ()
    | Noeud(r, g, d) -> (if p <> -1 then (ga.(r) <- p::ga.(r);
                                          ga.(p) <- r::ga.(p));
                         aux r g;
                         aux r d) in
  aux (-1) a; (* -1 pour indiquer que la racine n'a pas de père *)
  ga;;
```

26. On peut numéroter les n sommets de l'arbre avec `numerotation`, le transformer en graphe avec `arbre_vers_graphe`, puis calculer son diamètre en utilisant la question 19.
`numerotation` et `arbre_vers_graphe` parcourent chaque sommet de l'arbre une fois en faisant un nombre constant d'itérations, donc sont en $O(n)$. Comme le nombre d'arête de l'arbre est $n-1$, d'après la réponse à la question 20, la méthode pour calculer le diamètre dans le graphe obtenu est en $O(n^2)$.
D'où la complexité totale $O(n) + O(n) + O(n^2) = \boxed{O(n^2)}$.

27. On note $\|C\|$ la longueur d'un chemin C. Si C est vide (c'est à dire qu'il ne contient aucun sommet) on définit $\|C\| = -1$. On pose aussi $h(\text{Feuille}) = 0$ (non défini par l'énoncé).
Soit $\mathcal{A}$ un arbre et C un chemin maximal de $G_{\mathcal{A}}$. Supposons que C passe par la racine de $\mathcal{A}$.
Soit C_g la partie de C dans $G_{\mathcal{A}_g}$ et $\overrightarrow{C_g}$ le chemin correspondant dans $\mathcal{A}_g$. Montrons que $\|C_g\| = h(\mathcal{A}_g) - 1$.
$\overrightarrow{C_g}$ est un plus long chemin de la racine de $\mathcal{A}_g$ à un noeud de $\mathcal{A}_g$ (sinon, on pourrait remplacer C_g dans C par un chemin plus long, ce qui contredirait la maximalité de C).
Donc $\|C_g\| = \|\overrightarrow{C_g}\| = h(\mathcal{A}_g) - 1$. Remarquons que cette formule reste vraie si $\mathcal{A}_g$ est une feuille.
On raisonne de même pour la partie C_d de C dans $G_{\mathcal{A}_d}$, d'où $\|C\| = \|C_g\| + 2 + \|C_d\| = \boxed{h(\mathcal{A}_g) + h(\mathcal{A}_d)}$ (on rajoute 2 pour les arêtes sortantes de la racine de $\mathcal{A}$).

28. Le diamètre est obtenu récursivement en remarquant qu'un chemin de longueur maximum est soit entièrement dans $\mathcal{A}_g$ (donc de longueur égale au diamètre de $\mathcal{A}_g$), soit entièrement dans $\mathcal{A}_d$ (donc de longueur égale au diamètre de $\mathcal{A}_d$) soit passe par la racine (donc de longueur $h(\mathcal{A}_g) + h(\mathcal{A}_d)$, d'après la question précédente).
On a besoin à la fois du diamètre et de la hauteur dans cette formule de récurrence, il est donc judicieux d'utiliser une fonction auxiliaire qui renvoie les deux informations:

```
let diam_arbre arb =
  let rec aux a = match a with (* renvoie (diamètre de a, hauteur de a) *)
    | Feuille -> (-1, 0)
    | Noeud(_, g, d) -> let dg, hg = aux g in
                        let dd, hd = aux d in
                        (max (max dg dd) (hg + hd), 1 + max hg hd) in
  fst (aux arb);;
```

On choisit de renvoyer $(-1, 0)$ pour un arbre réduit à une feuille en accord avec les conventions utilisées dans la réponse à la question 27.
`aux a` effectue un appel récursif pour chaque noeud de `a` et chacun de ces appels effectue un nombre constant d'opérations (en dehors des appels récursifs) donc la complexité de cette fonction est bien linéaire en le nombre de noeuds.
Remarque: ce ne serait pas le cas si on appelait une fonction recalculant la hauteur à chaque fois (on aurait alors une complexité quadratique).