OPTION INFORMATIQUE
Devoir surveillé 2

Toutes les fonctions sont à écrire en OCaml. On attachera le plus grand soin à leur lisibilité. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée.

1 Algorithmes de tri

On dit qu'une liste ou un tableau est trié si ses éléments sont rangés dans l'ordre croissant.

1.1 Test de tri

1. Écrire une fonction `liste_triee` prenant en argument une liste et renvoyant `true` si elle est triée et `false` sinon.
2. Écrire une fonction `tableau_trie` prenant en argument un tableau et renvoyant `true` s'il est trié et `false` sinon.
3. Déterminer la complexité des deux fonctions précédentes.

1.2 Tri par insertion

1. (a) Écrire une fonction `insertion : 'a list -> 'a -> 'a list` prenant en argument une liste l supposée triée et un élément x et renvoyant une liste triée composée des éléments de l ainsi que x.
   (b) En déduire une fonction `tri_insertion_liste : 'a list -> 'a list` prenant en argument une liste et renvoyant une liste triée composée des mêmes éléments.
   (c) Déterminer la complexité de cette fonction. On distinguera meilleur et pire cas.
2. (a) Écrire une procédure `tri_insertion_tableau : 'a array -> unit` prenant en argument un tableau et le modifiant pour le trier par insertion.
   (b) Déterminer la complexité de cette fonction. On distinguera meilleur et pire cas.
   (c) Démontrer la terminaison et la correction de cette fonction. Chaque boucle devra être analysée à l'aide d'un invariant de boucle.

1.3 Tri fusion

Le tri fusion utilise la stratégie diviser pour régner : on va diviser la liste à trier en deux parties, les trier puis combiner les résultats.

1. Écrire une fonction `diviser : 'a list -> 'a list * 'a list` prenant en argument une liste l et renvoyant deux listes de longueurs égales ou quasi-égales et dont la réunion contient tous les éléments de l.
2. Écrire une fonction `fusion : 'a list -> 'a list -> 'a list` prenant en argument deux listes $l1, l2$ supposées triées et renvoyant une liste triée formée des éléments de $l1$ et de $l2$. La complexité devra être linéaire en $O(n)$ où $n =$ `len(l1)` + `len(l2)`, ce qu'on justifiera.
3. Écrire une fonction `tri_fusion : 'a list -> 'a list` prenant en argument une liste l et renvoyant une liste triée contenant les mêmes éléments, en utilisant les deux fonctions précédentes.
4. Appliquer le théorème maître pour déterminer la complexité de `tri_fusion`.

1.4 Tri rapide sur liste

Le tri rapide utilise la stratégie suivante : on extrait de la liste à trier son premier élément, on répartit les autres éléments selon qu'ils sont inférieurs ou supérieurs à ce premier élément, on trie récursivement ces deux listes et on en déduit le résultat.

1. Écrire une fonction `repartition : 'a list -> 'a -> 'a list * 'a list` prenant en argument une liste l et un élément x, et renvoyant un couple de listes $r1, r2$ où $r1$ est formée des éléments de l strictement inférieurs à x, et $r2$ des éléments de l supérieurs ou égaux à x.
2. Écrire une fonction `concatenation : 'a list -> 'a list -> 'a list` prenant en argument deux listes et renvoyant leur concaténation.
3. Écrire une fonction `tri_rapide_liste : 'a list -> 'a list` prenant en argument une liste l et renvoyant une liste triée contenant les mêmes éléments, en utilisant les deux fonctions précédentes.
4. Déterminer la complexité de `repartition` et `concatenation`.
5. Déterminer la complexité de `tri_rapide_liste`, dans le cas où les répartitions successives produisent toujours des listes de longueurs égales.
6. Déterminer la complexité de `tri_rapide_liste`, dans le cas où les répartitions successives sont toujours maximalement déséquilibrées. Donner un exemple de liste pour laquelle ce cas est atteint.
7. Démontrer la terminaison et la correction de `tri_rapide_liste` (on admettra la terminaison et la correction de `repartition` et `concatenation`).

1.5 Tri rapide en place

On rappelle qu'un tri est en place s'il permute les éléments directement dans le tableau argument, en utilisant une quantité de mémoire en $O(1)$ par ailleurs. L'objectif de cette partie est d'implémenter le tri rapide en place.

1. Écrire une fonction `repartition_en_place : 'a array -> int -> int -> int` prenant en argument un tableau `t` et deux indices g et d, prenant `t.(d)` comme pivot et permutant les éléments de `t` entre les indices g et d de façon à ce que le pivot ait à sa gauche des éléments inférieurs et à sa droite des éléments supérieurs ou égaux. La fonction renverra par ailleurs l'indice du pivot après ces permutations. La fonction devra avoir une complexité temportelle en $O(d-g)$, ce qu'on justifiera.
2. En déduire une fonction `tri_rapide_en_place : 'a array -> unit` implémentant le tri rapide en place.

1.6 Tri par comptage

Le tri par comptage ne fonctionne que sur un tableau d'entiers, de valeur minimum m et de valeur maximum M. Son principe est de créer un tableau de longueur $M-m+1$ tel que la case d'indice i contienne le nombre d'occurrences de la valeur $m+i$ dans le tableau à trier.

1. Écrire une fonction `tri_comptage : int array -> unit` implémentant cet algorithme de tri.
2. Déterminer la complexité de cet algorithme, en fonction de la longueur n du tableau à trier et de $e = M-m+1$.
3. Ce tri est-il en place ? Justifier.
4. Donner sans justifier pour chaque boucle un invariant de boucle garantissant la correction de cette fonction.

2 Arbres et arités

On considère le type d'arbres suivant :

```
type 'a arbre =
  V
| N of 'a * 'a arbre list;;
```

1. Définir en Caml deux arbres ayant `[1; 2; 3]` comme noeuds visités dans l'ordre du parcours préfixe, et des parcours postfixes différents. Préciser le parcours postfixe de chaque arbre.
2. Écrire une fonction `somme : int arbre -> int` prenant en argument un arbre contenant des entiers, et renvoyant la somme de ses noeuds.
3. On définit l'arité d'un noeud de l'arbre comme son nombre de fils **non vides**. Écrire une fonction `arite_max : 'a arbre -> int` prenant en argument un arbre, et renvoyant l'arité maximale de ses noeuds. On conviendra que l'arbre vide a pour arité maximale 0.
4. On dit qu'un arbre est entier si tous ses noeuds qui ne sont pas d'arité 0 ont la même arité. Écrire une fonction `est_entier : 'a arbre -> bool` testant si un arbre est entier.

3 Problème de l'arrêt

Si f est une fonction et x un objet quelconque, on dit que f termine sur x si l'évaluation de l'appel $f\ x$ renvoie un résultat au bout d'un temps fini, ou est interrompue à la suite d'une erreur, par exemple si f et x n'ont pas des types compatibles.

1. (a) Déterminer les entiers pour lesquels la fonction suivante termine :

```
let rec f1 n =
    match n with
         0 -> 0
        |_ -> n + f1(n-2)
```

   (b) Démontrer la terminaison de cette fonction sur cet ensemble d'entiers.

2. Mêmes questions pour la fonction :

```
let f2 n =
    let i = ref n in
    while !i != 10 do
        print_int(!i);
        incr(i);
    done;;
```

3. Écrire une fonction qui ne termine sur aucun entier.
4. L'objectif de cette question est d'examiner la possibilité de décider automatiquement si une fonction f termine sur une entrée x, en analysant le code de f.

   On suppose l'existence d'une fonction `arret` prenant en arguments une chaîne de caractères `code_f` correspondant au code d'une fonction f, ainsi qu'un objet de type quelconque x, et renvoyant `true` si f termine sur x et `false` sinon. On notera en particulier que la fonction `arret` termine dans les deux cas.

   (a) Donner le type de la fonction `arret`.

   (b) Écrire une fonction `bizarre` prenant en argument une chaîne de caractères `code_f` correspondant au code d'une fonction f, ainsi qu'un objet de type quelconque x, telle que `bizarre` termine sur `code_f` et x si et seulement si f ne termine pas sur x. Le résultat renvoyé par `bizarre` quand elle termine n'a aucune importance.

   (c) Écrire une fonction `paradoxe` prenant en argument une chaîne de caractères `code_f` correspondant au code d'une fonction f, et terminant sur `code_f` si et seulement si f ne termine pas sur `code_f`.

   (d) En utilisant `code_paradoxe` le code de la fonction `paradoxe`, exhiber une contradiction et conclure.

4 Produit matriciel

L'objectif de cette partie est de calculer le produit matriciel de deux matrices, représentées en Caml par des tableaux de tableaux de flottants (`float array array`)).
Dans la suite, pour créer une nouvelle matrice de dimension $n \times m$ dont les valeurs sont initialement toutes nulles, on utilisera l'expression `Array.make_matrix n m 0.`

4.1 Algorithme naïf

1. Écrire une fonction `produit` prenant en argument deux matrices et renvoyant leur produit, qu'on calculera en suivant la définition du produit matriciel. On supposera que les deux matrices sont de dimensions compatibles pour le produit.
2. Déterminer la complexité de la fonction précédente, en fonction des dimensions des matrices.

4.2 Diviser pour régner naïf

L'application de la méthode diviser pour régner au calcul matriciel repose sur le principe du calcul par blocs : en notant pour une matrice A, de dimension $n \times n$ avec n une puissance de 2, $A_{0,0}$ son quart supérieur gauche, $A_{0,1}$ son quart supérieur droit , $A_{1,0}$ son quart inférieur gauche et $A_{1,1}$ son quart inférieur droit, alors le produit $C = AB$ vérifie

- $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$
- $C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1}$
- $C_{1,0} = A_{1,0}B_{0,0} + A_{1,1}B_{1,0}$
- $C_{1,1} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1}$

1. Écrire une fonction `découpage` prenant une telle matrice et renvoyant le quadruplet formé de ses quatre quarts.
2. Écrire une fonction `recollage` prenant en argument quatre matrices `a00`, `a01`, `a10` et `a11` et renvoyant la matrice `a` dont les matrices arguments forment le découpage en quarts.
3. Écrire une fonction `somme` renvoyant la somme de deux matrices de même dimension.
4. Déterminer la complexité des trois fonctions précédentes.
5. Écrire une fonction `produit_dpr` prenant en argument deux matrices carrées dont la taille est une puissance de 2, et calculant leur produit matriciel en appliquant la méthode diviser pour régner et le produit par bloc.
6. Appliquer le théorème maître pour déterminer la complexité de la fonction précédente. Comment se compare-t-elle à la complexité de l'algorithme naïf ?

4.3 Algorithme de Strassen

L'algorithme de Strassen est une optimisation de l'idée précédente, se basant sur le calcul des sept matrices suivantes :

- $M_1 = (A_{0,0} + A_{1,1})(B_{0,0} + B_{1,1})$
- $M_2 = (A_{1,0} + A_{1,1})B_{0,0}$
- $M_3 = A_{0,0}(B_{0,1} - B_{1,1})$
- $M_4 = A_{1,1}(B_{1,0} - B_{0,0})$
- $M_5 = (A_{0,0} + A_{0,1})B_{1,1}$
- $M_6 = (A_{1,0} - A_{0,0})(B_{0,0} + B_{0,1})$
- $M_7 = (A_{0,1} - A_{1,1})(B_{1,0} + B_{1,1})$

Ces sept matrices permettent par combinaisons linéaires de calculer $C_{0,0}, C_{0,1}, C_{1,0}$ et $C_{1,1}$. Par exemple, on observe que $C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0} = M_1 + M_4 - M_5 + M_7$.

1. Exprimer de même $C_{0,1}, C_{1,0}$ et $C_{1,1}$ comme combinaisons linéaires des M_i.
2. En déduire une fonction `strassen` calculant le produit matriciel de deux matrices carrées de dimension une puissance de deux.
3. Déterminer la complexité de cet algorithme, et la comparer à celle des méthodes précédentes.