Devoir surveillé n°2 - Partie papier

mardi 5 mai - durée : 45min

L'usage de tout document (y compris en ligne), de la calculatrice ou d'un ordinateur est interdit.
Il s'agit d'un travail individuel : tout travail en groupe est proscrit et sera considéré comme une tentative de fraude.

Il sera tenu compte de la rigueur, du soin et de la rédaction dans la notation. Les copies illisibles ou mal présentées seront pénalisées.
On pourra admettre **en l'énonçant clairement** le résultat d'une question afin de l'utiliser par la suite.
On pourra de même utiliser une fonction définie par une question précédente.

Consignes pour cette partie du devoir

Pour rendre votre travail

- Rendre une version scannée de votre copie, en **un seul fichier au format pdf**, intitulé `ds2_Nom_Prenom.pdf`, en la déposant à l'adresse suivante :
  `https://maths.mlong.fr/nextcloud/index.php/s/WyNyt7gNkzA8eeL`

Exercice 1 - Piles et files

On considère la suite d'opérations suivantes :

- Ajouter dans cet ordre les éléments 1, 2, 3, 4 et 5.
- Supprimer et afficher successivement deux éléments.
- Ajouter dans cet ordre les éléments 10, 11 et 12.
- Supprimer et afficher successivement cinq éléments.

Quel est l'affichage obtenu :

1. si on utilise une pile ?
2. si on utilise une file ?

Exercice 2 - Autour des arbres binaires

On considère des arbres binaires étiquetés par des entiers. Pour cela, on utilise le type suivant :

```
type arbre =
  | Vide
  | Noeud of int * arbre * arbre
;;
```

1. (a) Écrire une fonction `mini : arbre -> int` qui prend en argument un arbre et qui renvoie le minimum de ses étiquettes.

   (b) Prouver que cette fonction termine.

2. On rappelle que la profondeur d'un nœud est sa distance par rapport à la racine (la profondeur de la racine est 0, celle des fils éventuels de la racine est 1, etc.).
   Écrire une fonction `noeuds_profondeur : arbre -> int -> int list` qui prend en argument un arbre et un entier p et qui renvoie la liste des nœuds de l'arbre dont la profondeur est p.
   On pourra utiliser l'opérateur `@` de concaténation sur les listes.

Exercice 3 - Nombre d'inversions d'un tableau

Étant donné un tableau d'entiers $t = [|t_0; t_1; \dots; t_{n-1}|]$, on appelle *inversion* de t tout couple $(i, j) \in [\![0, n-1]\!]^2$ tel que $i < j$ et $t_i > t_j$.
Par exemple, le tableau $[|2; 4; 1; 5; 3|]$ possède quatre inversions $(0, 2)$, $(1, 2)$, $(1, 4)$ et $(3, 4)$.

1. Écrire une fonction `nb_inversions : 'a array -> int` qui prend en argument un tableau et qui renvoie son nombre d'inversions en examinant de manière exhaustive tous les couples $(i, j) \in [\![0, n-1]\!]^2$ tel que $i < j$.
2. Montrer que la complexité de cette fonction est un $O(n^2)$.

On souhaite améliorer cette complexité en adoptant une stratégie « diviser pour régner ».
En effet, si (i, j) est une inversion d'un tableau t, et si on partage t en deux parties t_1 et t_2, alors

- ou bien les indices i et j sont tous les deux dans la première moitié du tableau ;
- ou bien i et j sont tous les deux dans la deuxième moitié du tableau ;
- ou bien i est dans la première moitié du tableau et j dans la deuxième.

Par conséquent, si on note k_1 le nombre d'inversions de t_1, k_2 le nombre d'inversions de t_2 et k_3 le nombre de couples (i_1, i_2) tels que $t_1.(i_1) > t_2.(i2)$, alors le nombre d'inversions de t est $k_1 + k_2 + k_3$.

3. On souhaite écrire une fonction

   `fusion : int array -> int array -> int array * int`

prenant en argument deux tableaux t_1 et t_2 supposés triés dans l'ordre croissant et renvoyant un tableau t trié dans l'ordre croissant, contenant les éléments de la concaténation de t_1 et t_2, ainsi que le nombre de couples (i_1, i_2) tels que $t_1.(i_1) > t_2.(i_2)$.
Recopier et compléter l'ébauche de fonction suivante, en prenant soin d'avoir une complexité linéaire en $n_1 + n_2$ où n_1 et n_2 sont les longueurs respectives de $t_1$et t_2.

```
let fusion t1 t2 =
  let n1 = Array.length t1 in
  let n2 = Array.length t2 in
  let nb_inv = ref 0 in
  if n1+n2 = 0
  then
    (* A compléter *)
  else
    let t = Array.make (n1+n2) (-1) in
    let i1 = ref 0 and i2 = ref 0 in
    (* Tests à compléter *)
    while !i1 < ... && !i2 < ... do
      (* On doit déterminer t.(!i1 + !i2) *)
      if t1.(!i1) <= t2.(!i2)
      then
        begin
          (* A compléter *)
        end
      else
        begin
          (* A compléter *)
        end
    done;
    (* Il peut rester des éléments non parcourus dans t1 ou dans t2,
 compléter... *)
    t, !nb_inv
;;
```

4. En déduire une fonction récursive

   `nb_inversions_tri : int array -> int array * int`

   qui prend en argument un tableau t et qui renvoie un tableau trié dans l'ordre croissant contenant les mêmes éléments que t ainsi que le nombre d'inversions de t.

   Pour séparer t en deux parties, on pourra utiliser la fonction
   `Array.sub : 'a array -> int -> int -> 'a array`,
   qui prend en argument un tableau t, un indice i et un nombre d'éléments k et qui renvoie la portion de k éléments du tableau t à partir de l'indice i compris).

   La complexité de cette fonction est linéaire par rapport à k.
5. Montrer que la complexité $C(n)$ dans le pire des cas pour un tableau de longueur n vérifie la relation

$$C(n) \leqslant \alpha n + \beta + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right)$$

   où α et β sont des constantes qu'on ne cherchera pas à déterminer.
6. Pour tout $p \in \mathbb{N}$, on pose $u_p = C(2^p)$. Montrer que $u_p = O(p2^p)$.
7. En supposant que C est croissante, que peut-on en déduire pour $C(n)$?
8. Comparer la complexité des deux algorithmes.