Devoir surveillé n°2

Durée : 2 heures, calculatrices et documents interdits

I. Écriture binaire et addition.

Soit a un entier naturel non nul ; a admet une unique écriture binaire $a = \sum_{i=0}^{n} 2^i a_i$ où $a_i \in \{0;1\}$ et $a_n = 1$. Le nombre a_i est appelé $i^{\text{ème}}$ bit de a. On représente l'entier positif non nul a par la liste de ses bits : $[a_0; a_1; \ldots; a_n]$, la liste vide représentant 0.

1) Écrire une fonction `successeur : int list -> int list`, qui prend en argument un entier positif a et qui renvoie $a+1$.

Dans la suite, on supposera que la fonction `successeur` termine et est correcte.

2) On souhaite désormais additionner deux entiers.

a) Artémis propose la fonction suivante :

```
let rec add a b =
  match a, b with
  | [], _ -> b
  | _, [] -> a
  | a0::qa, 0::qb -> a0::(add qa qb)
  | 0::qa, b0::qb -> b0::(add qa qb)
  | _::qa, _::qb -> 0::successeur (add qa qb)
;;
```

Montrer que sa fonction termine et renvoie le bon résultat.

b) Apollon propose une autre fonction :

```
let rec add a b =
  match a with
  | [] -> b
  | a0::qa ->
    match b with
    | [] -> a
    | 0::qb -> a0::(add qa qb)
    | _::qb -> successeur (add a (0::qb))
;;
```

Montrer que sa fonction termine.

II. Énumération des parties de $[\![1, n]\!]$.

Pour tout $n \in \mathbb{N}^*$ on note E_n l'ensemble $[\![1, n]\!]$ des entiers de 1 à n, et on pose $E_0 = \varnothing$.
Une partie A de E_n est représentée par une liste $[a_1; \ldots; a_p]$ telle que $A = \{a_1, \ldots, a_p\}$ et $a_1 < a_2 < \cdots < a_p$.

1) Écrire une fonction `ensemble` qui prend en argument un entier n et qui renvoie la liste correspondant à E_n (avec les contraintes données ci-dessus).

2) **a)** Soit A une partie de E_n représentée par une liste ℓ et soit $x \in E_n$. Écrire des fonctions `add` et `sub` prenant en argument ℓ et x et qui renvoient respectivement les listes représentant les parties $A \cup \{x\}$ et $A \setminus \{x\}$.

b) Écrire des fonctions `reunion` et `intersection` prenant en argument deux listes ℓ_1 et ℓ_2 représentant deux parties A_1 et A_2 de E_n et renvoyant la liste correspondant respectivement à la réunion et à l'intersection de A_1 et A_2.

3) **a)** Écrire une fonction `parties` prenant en argument deux entiers n et p et revoyant la liste de toutes les parties à p éléments de l'ensemble E_n.
Par exemple, lorsque $n = 2$ et $p = 1$, la fonction reverra `[[1];[2]]` ; lorsque $n = 2$ et $p = 0$, la fonction renverra `[[]]`.
On pourra remarquer que l'ensemble des parties à p éléments se partitionne en l'ensemble des parties à p éléments contenant n et l'ensemble des parties à p éléments ne contenant pas n.

b) Détailler l'exécution de votre fonction pour l'appel `parties 3 2`.

III. Recherche des absents.

Une annexe en fin de devoir rappelle quelques éléments de programmation autour de la structure de tableau.
Dans une classe de n élèves, les élèves sont numérotés de 0 à $n-1$. Un professeur souhaite faire l'appel, c'est à dire déterminer quels élèves sont absents.

Partie A

1) Ecrire une fonction `mini : int list -> (int * int list)` qui prend en argument une liste non vide d'entiers distincts, et renvoie le plus petit élément de cette liste, ainsi que la liste de départ privée de cet élément (pas forcément dans l'ordre initial).

2) En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de comparaisons de la fonction précédente ?

3) En utilisant la fonction `mini`, écrire une fonction `absents : int list` $\to$ `int` $\to$ `int list` qui, étant donné une liste non vide d'entiers distincts et n, renvoie, dans un ordre quelconque, la liste des entiers de $[0; n-1]$ qui n'y sont pas.

4) En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de comparaisons (en fonction de n et k) de la fonction précédente ?

Partie B

Dans cette partie, une salle de classe pour n élèves est décrite par la donnée d'un tableau à n entrées. Si `tab` est un tel tableau et `i` un entier de $[0; n-1]$, alors `tab.(i)` donne le numéro de l'élève assis à la place i (ou -1 si cette place est vide).

5) Ecrire une fonction `asseoir : int list -> int -> int array`, qui prend en argument une liste non vide d'entiers distincts et un entier n, et renvoie un tableau représentant une salle de classe pour n élèves où chaque élève de la liste à été assis à la place numérotée par son propre numéro. Les entiers supérieurs ou égaux à n seront ignorés.

6) En déduire une fonction `absents2 : int list -> int -> int list` qui étant donné une liste non vide d'entiers distincts et un entier n, renvoie la liste des entiers de $[0; n-1]$ qui n'y sont pas. Les entiers supérieurs ou égaux à n seront ignorés.

7) En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de lectures et d'écritures dans un tableau (en fonction de n et k) de la fonction précédente ?

IV. Inversions d'un tableau.

Une annexe en fin de devoir rappelle quelques éléments de programmation sur les tableaux.
Étant donné un tableau d'entiers $t = [|t_0; t_1; \ldots; t_{n-1}|]$, 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
          (* A compléter *)
      else
          (* A compléter *)
    done;
    (* Il peut rester des éléments non parcourus, compléter... *)
    (* On renvoie le tableau t et le nombre de couples (i1,i2) ↙
       tels que t1.(i1) > t2.(i2) *)
    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 $u_p = \frac{C(2^p)}{2^p}$. Montrer que $u_p = O(p)$.

7) En supposant que C est croissante, que peut-on en déduire pour $C(n)$?

8) Comparer la complexité des deux algorithmes.

Annexe : Rappels sur les tableaux.

— La fonction `Array.length : 'a array -> int` permet d'obtenir la longueur d'un tableau.
— Si `t` est un tableau à n éléments, ceux-ci sont indexés de 0 à $n-1$ et pour tout $i \in [\![0, n-1]\!]$, `t.(i)` permet d'obtenir la valeur de l'élément d'indice i de `t`.
— La valeur de l'élément d'indice i de `t` peut être modifiée avec `t.(i) <-` e où e est une expression.
— La fonction `Array.make : int -> 'a -> 'a array` permet de créer un tableau d'une longueur donnée avec pour élément par défaut le second argument de la fonction.

— FIN —