DS n°1 d'informatique (3 heures 30)

Ce devoir est composé de 3 problèmes indépendants.
Une attention particulière sera accordée à la présentation des copies, spécialement la lisibilité des codes.
Les programmes qui ne sont pas évidents devront être expliqués ou commentés.

Problème n°1 : coupe maximale dans une somme

Soit T un tableau de nombres entiers relatifs dont les cases sont numérotées de 0 à $n-1$. Si i et j sont deux entiers naturels tels que $0 \leqslant i \leqslant j \leqslant n-1$, on appelle sous-somme de T de la case i à la case j, et on note $\sigma_{i \to j}$, la somme des valeurs consécutives de T entre les cases i et j, c'est-à-dire $\sigma_{i \to j}(T) = T[i] + T[i+1] + \cdots + T[j]$.
On cherche la valeur maximale de toutes les sous-sommes de T, appelée coupe maximale.
Par exemple, pour le tableau $T =$`[|2;5;-8;6;5;-1;2;-9;3;4|]`, la coupe maximale est $6+5-1+2 = 12$.
On se propose d'étudier deux algorithmes de résolution de ce problème.

1. (a) Ecrire une fonction `maxsomme : int array -> int` calculant à l'aide d'une double boucle la valeur d'une coupe maximale. On garantira que la taille des résultats intermédiaires stockés soient en $\mathcal{O}(1)$.
   (b) Préciser la complexité de cet algorithme.
2. On souhaite proposer une méthode basée sur le principe "diviser pour régner". Pour chaque zone du tableau initial comprise entre les indices i et j, on va calculer récursivement 4 valeurs, en partageant la zone étudiée en 2 parties égales.
   (a) Ecrire une fonction `maxsomme2 : int array -> int` calculant une sous-somme maximale, utilisant une fonction récursive sur les indices qui, à partir de la zone du tableau comprise entre les indices i et j, calcule le quadruplet (s, msg, ms, msd) telle que :
   - s est la somme $\sigma_{i \to j}(T)$,
   - msg est le maximum des sous-sommes $\sigma_{i \to k}(T)$ pour $k \in [\![i\,,j]\!]$,
   - ms est le maximum des sous-sommes $\sigma_{k \to l}(T)$ pour k et l tels que $i \leqslant k \leqslant l \leqslant j$,
   - msd est le maximum des sous-sommes $\sigma_{k \to j}(T)$ pour $k \in [\![i\,,j]\!]$.

   On garantira que la complexité de la fonction `maxsomme2` vérifie l'équation suivante :
   $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(1)$.
   On expliquera comment, en coupant la zone du tableau au milieu, on déduit le quadruplet cherché de la connaissance du quadruplet pour les parties gauche et droite.
   (b) En déduire, preuve à l'appui dans le cas où n est une puissance de 2, la complexité de cet algorithme.

Problème n°2 : recherche d'un élément majoritaire

On étudie des tableaux remplis d'entiers naturels non nuls. On souhaite savoir si un élément d'un tel tableau est majoritaire, c'est-à-dire s'il apparait strictement plus de $\frac{n}{2}$ fois dans un tableau de longueur n.
Par exemple, l'élément 4 est majoritaire pour le tableau `[|2;4;5;1;4;4;4|]`, alors que le tableau `[|1;2;3;4;6;2;3;3|]` n'admet pas d'élément majoritaire.
L'objectif est de coder une fonction `majoritaire : int array -> bool * int` telle que `majoritaire t` renvoie `(true,x)` si `x` est majoritaire dans `t` et `(false,0)` si `t` ne possède pas d'élément majoritaire.
Pour des raisons de simplicité, on se limite à des tableaux dont la taille est une puissance de 2.

1. (a) Ecrire une fonction `occurrences : int -> int array -> int -> int` telle que `occurrences x t i j` calcule le nombre d'occurrences de `x` dans le tableau `t` entre les indices `i` et `j`.

(b) Indiquer la complexité de cet algorithme en fonction de `i` et `j`.

(c) En déduire une fonction `majoritaire : int array -> bool * int` répondant au problème.

(d) Indiquer la complexité de cet algorithme.

2. On se propose d'utiliser la stratégie "diviser pour régner" pour déterminer un élément majoritaire d'un tableau.

(a) Soit a une zone de tableau comprise entre les indices `i` et `j`, a_1 sa première moitié et a_2 sa seconde moitié. Montrer que si x est majoritaire dans a, alors il est majoritaire dans a_1 ou dans a_2.

(b) Ecrire une fonction `majoritaire2 : int array -> bool * int` répondant au problème suivant la stratégie "diviser pour régner". Cette fonction utilisera une fonction auxiliaire récursive `majo i j` renvoyant `(true,x)` si l'élément `x` est majoritaire dans la zone de tableau comprise entre les indices `i` et `j` et `(false,0)` en l'absence d'élément majoritaire. Cette fonction effectuera 2 appels récursifs sur les deux moitiés de la zone comprise entre `i` et `j` et filtrera suivant les 4 couples de booléens obtenus.

(c) Justifier que cet algorithme vérifie l'équation de complexité $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(n)$. En déduire un ordre de grandeur de sa complexité en fonction de n.

3. On se propose d'améliorer la stratégie "diviser pour régner" pour déterminer un élément majoritaire d'un tableau.

Soit T un tableau de longueur n. On dit que le nombre entier x est un ***postulant*** pour la valeur c_x du tableau T si c_x est un entier strictement supérieur à $\frac{n}{2}$ tel que x apparait au plus (au sens large) c_x fois dans T et tout entier y distinct de x, apparait au plus (au sens large) $n - c_x$ fois dans T.

Par exemple, $x = 3$ est un postulant pour la valeur $c_x = 5$ du tableau `[|1;2;3;4;3;2;3;3|]`

On dit que le nombre entier x est un postulant du tableau T s'il existe un nombre entier $c_x > \frac{n}{2}$ tel que x est un postulant pour la valeur c_x du tableau T.

(a) Démontrer que si a est majoritaire pour le tableau T, alors a est un postulant de T.

(b) Démontrer que si a est un postulant de T, alors aucun autre élément de T ne peut être majoritaire.

(c) Donner un exemple de tableau qui contient un postulant sans élément majoritaire et un exemple de tableau n'ayant aucun postulant.

(d) Soit T un tableau de longueur un entier pair n. On note T_G le tableau de longueur $\frac{n}{2}$ formé par les $\frac{n}{2}$ premières cases de T et T_D le tableau de longueur $\frac{n}{2}$ formé par les $\frac{n}{2}$ dernières cases de T.

i. Soit x un postulant de T_G pour la valeur c_x. On suppose que le tableau T_D n'a pas d'élément majoritaire. Montrer que x est un postulant de T pour la valeur $c_x + \lfloor \frac{n}{4} \rfloor$.

ii. Soit x un postulant commun à T_G et T_D. Montrer que x est un postulant de T et indiquer pour quelle valeur.

iii. Soient x un postulant de T_G pour la valeur c_x et y un postulant de T_D distinct de x pour la valeur c_y.

A. Si $c_x = c_y$, prouver que T n'admet pas d'élément majoritaire.

B. Si $c_x > c_y$, montrer que x est un postulant de T pour la valeur $\frac{n}{2} + c_x - c_y$.

(e) Ecrire une fonction `postulant : int array -> int -> int -> bool * int` telle que `postulant t i j` renvoie `(true,x,cx)` si `x` est un postulant pour la valeur `cx` dans la zone comprise entre `i` et `j` et `(false,0,0)` si cette zone n'admet pas d'élément majoritaire.

(f) Montrer que la complexité pour une zone de longueur n de cette fonction suit la relation $T(n) = 2T(\frac{n}{2}) + \mathcal{O}(1)$. En déduire qu'elle est linéaire.

(g) En déduire une fonction `majoritaire3` répondant au problème.

(h) Montrer que la complexité de cette fonction est linéaire.

Problème n°3 : opérations sur les entiers longs

On souhaite manipuler des entiers relatifs dépassant les capacités de OCaml.
On se donne pour cela une base de calcul, par exemple `base = 10000`. La valeur de la base importe peu, du moment qu'elle est paire, supérieure ou égale à 2 et que son double n'excède pas le plus grand entier machine.
Un entier naturel de précision arbitraire est alors représenté par la liste de ses chiffres en base `base`, les chiffres les moins significatifs étant en tête de liste. Ainsi la liste `[1;2;3]` représente l'entier $1+2\times\texttt{base}+3\times\texttt{base}^2$.
On définit le type `nat` suivant : `type nat = int list ;;`
Dans la suite, on garantira l'invariant suivant sur le type `nat` :
- tout élément de la liste est compris entre 0 et $\texttt{base}-1$ au sens large.
- le dernier élément de la liste, lorsqu'il existe, n'est pas nul.

On notera que l'entier 0 est représenté par la liste vide.

Les programmes écrits dans ce problème devront être récursifs. L'utilisation des références est proscrite.

1. Définir une fonction `cons_nat : int -> nat -> nat` qui prend en argument un chiffre c ($0 \leqslant c < \texttt{base}$) et un grand entier n, et qui renvoie le grand entier $c+\texttt{base}\times n$.
2. Définir une fonction `add_nat : nat -> nat -> nat` qui calcule la somme de deux grands entiers. *On pourra commencer par écrire une fonction prenant également une retenue en argument et appliquer l'algorithme traditionnel enseigné à l'école primaire.*
3. Définir une fonction `cmp_nat : nat -> nat -> int` qui prend en arguments deux grands entiers n_1 et n_2, et qui renvoie -1 si $n_1 < n_2$, 1 si $n_1 > n_2$ et 0 si $n_1 = n_2$.
4. Définir une fonction `sous_nat : nat -> nat -> nat` qui prend en arguments deux grands entiers n_1 et n_2, et qui calcule la différence $n_1 - n_2$ en supposant $n_1 \geqslant n_2$.
   On suivra la même indication que pour l'addition.
5. Définir une fonction `div2_nat : nat -> nat * int` qui prend en argument un grand entier n et qui calcule le quotient et le reste de la division euclidienne de n par 2. Le quotient est un grand entier et le reste un entier valant 0 ou 1. On rappelle que la constante `base` est paire.

   A partir de ces grands entiers naturels, on va maintenant construire de grands entiers relatifs. Pour cela, on introduit le type enregistrement `z` suivant, où le champ `signe` contient le signe de l'entier relatif, à savoir 1 ou -1, et le champ `nat` sa valeur absolue : `type z = {signe : int ; nat : nat}` On notera que 0 admet deux représentations, ce qui n'est pas gênant pour la suite.
6. Définir une fonction `neg_z : z -> z` qui calcule la négation d'un grand entier relatif.
7. Définir une fonction `add_z : z -> z -> z` qui calcule la somme de deux grands entiers relatifs.
8. Définir une fonction `mul_puiss2_z : int -> z -> z` qui prend en arguments un entier machine $p \geqslant 0$ et un grand entier relatif `z`, et qui renvoie le grand entier relatif $2^p z$.
   On se contentera d'une solution simple, sans viser particulièrement l'efficacité.
9. Définir une fonction `decomp_puiss2_z : z -> z * int` qui prend en argument un grand entier relatif z non nul et qui renvoie un grand entier relatif u impair et un entier machine p tels que $z = 2^p u$. Cette fonction calcule donc la plus grande puissance de 2 qui divise z et renvoie la décomposition correspondante.
   Comme avant, on visera la simplicité et on supposera que z est tel que p est bien représentable par un entier machine.