Devoir surveillé n°2

Durée : 2 heures, calculatrices et documents interdits

Pour répondre à une question vous pouvez à tout moment utiliser le programme d'une question antérieure, même si vous ne l'avez pas écrit.

1 Fonctions sur les listes

1) Écrire une fonction `appartient` qui teste l'appartenance d'un élément à une liste. Donner le type de cette fonction (on pourra utiliser le type `'a` pour symboliser n'importe quel type).
2) Écrire une fonction `supprime` qui supprime toutes les occurrences d'un élément dans une liste (on renverra une nouvelle liste). Par exemple `supprime 2 [1;2;3;2;4]` renverra `[1;3;4]`. Donner le type de cette fonction.
3) Écrire une fonction `ajoute` d'ajout d'un élément dans une liste sans redondance (si l'élément appartient déjà à la liste, on renvoie la liste sans la modifier). Donner le type de cette fonction.
4) En utilisant la fonction `ajoute`, coder une fonction `union` telle que `union l1 l2`, où `l1` et `l2` sont deux listes d'éléments sans doublon dans un ordre arbitraire, renvoie une liste sans doublon contenant l'union des éléments des deux listes, dans un ordre arbitraire. Par exemple `union [1;2;3;4] [2;4;6;8]` renvoie une liste contenant dans un ordre arbitraire `1;2;3;4;6;8` sans doublon.
5) Donner la complexité de la fonction `ajoute` en fonction des longueurs `n1` et `n2` des deux listes d'entrée.
6) En utilisant la fonction `union`, coder une fonction `fusion` telle que `fusion l`, où `l` est une liste de listes d'éléments, chacune de ces listes étant sans doublon, renvoie une liste de tous les éléments contenus dans au moins une des listes de la liste `l`, sans doublon et dans un ordre arbitraire. Par exemple `fusion [[1;2] ; [2;4] ; [4;8]]` renvoie une liste contenant dans un ordre quelconque les éléments `1;2;4;8` sans doublon.
7) Donner sa complexité en fonction des longueurs `ni` de chacune des listes contenues dans la liste `l`.
8) Coder une fonction `produit` telle que `produit l1 l2` renvoie une liste de tous les couples `(x, y)` avec `x` un élément de `l1` et `y` un élément de `l2`. On supposera les listes `l1` et `l2` sans doublon. La liste résultante doit avoir pour longueur le produit des longueurs des deux listes. Par exemple `produit [1;2] [3;4]` renvoie la liste contenant les couples `(1,3);(1,4);(2,3);(2,4)` dans un ordre arbitraire sans doublon.
9) Donner la complexité de la fonction `produit` en fonction des longueurs `n1` et `n2` des deux listes d'entrée.

2 Récursivité et accumulateurs

1) Écrire une fonction récursive `fact : int -> int` calculant la factorielle d'un entier $n \in \mathbb{N}$.

2) Écrire une fonction récursive terminale `fact2 : int -> int` calculant la factorielle d'un entier $n \in \mathbb{N}$. On pourra utiliser une fonction auxiliaire `aux : int -> int -> int` telle que `aux n acc` renvoie $(n!) \times$acc.
3) Écrire une fonction récursive terminale avec accumulateur(s) `fibo : int -> int` renvoyant le terme d'indice n de la suite de Fibonacci vérifiant : $f_0 = f_1 = 1$ et $\forall n \in \mathbb{N}$, $f_{n+2} = f_{n+1} + f_n$.

3 Représentation de polynômes

Soit $n \in \mathbb{N}^*$. On représente un polynôme $P = \sum_{k=0}^{n-1} a_k X^k$ de degré au plus $n-1$ par un tableau de ses coefficients $[|a_0; ...; a_{n-1}|]$. Par exemple le polynôme $1 + 2X^2$ pourra être représenté par le tableau `[|1;0;2|]` mais aussi par le tableau `[|1;0;2;0;0;0|]`.

On supposera dans cet exercice que tous les polynômes sont à coefficients entiers.

1) Écrire une fonction `degre : int array -> int` renvoyant le degré d'un polynôme représenté dans un tableau. On conviendra que le degré du polynôme nul est -1.
2) Écrire une fonction `somme : int array -> int array -> int array` renvoyant la somme de deux polynômes représentés par des tableaux. On ne supposera pas ces tableaux de même longueur. Ainsi `somme [|1;0;2|] [|0;1;2;3;4|]` renverra `[|1;1;4;3;4|]`.
3) Écrire une fonction naïve `produit : int array -> int array -> int array` renvoyant le produit de deux polynômes représentés par des tableaux. On ne supposera pas ces tableaux de même longueur. Ainsi `produit [|1;0;2|] [|2;-1|]` renverra `[|2;-1;4;-2|]`.
4) Écrire une fonction `composition : int array -> int array -> int array` renvoyant la composition de deux polynômes représentés par des tableaux. Si P_1 et P_2 sont représentés par les tableaux `t1` et `t2`, `composition t1 t2` renverra un tableau représentant $P1 \circ P2$. Ainsi `composition [|1;0;2|] [|2;-1|]` renverra `[|9;-8;2|]`.

4 Résultats d'élection

On souhaite analyser les résultats de sondages concernant une élection. Les candidats à l'élection sont numérotés de 0 à $k-1$. Les résultats de chaque sondage sont stockés dans un tableau. Si le sondage a recueilli N réponses, le tableau comporte N cases, une pour chaque réponse : la i-ème case du tableau contient le numéro du candidat proposé par la i-ème personne sondée. On a ainsi un tableau T de longueur N qui contient des entiers naturels entre 0et $k-1$. Le sondage donne le candidat numéroté i élu si le nombre i est dans strictement plus de $N/2$ cases de T. Par exemple, un sondage correspondant au tableau `t1 = [|2;4;5;0;4;4;4|]` donne le candidat 4 élu. Mais un tableau ne donne pas toujours un élu, par exemple le tableau `t2 = [|1;2;3;4;6;2;3;3|]`. On propose d'étudier deux stratégies pour effectuer le comptage des voix (dont une utilisant le principe diviser pour régner).

1) Écrire une fonction `nb : int array -> int -> int` telle que si `a` est un entier naturel et `tab` un tableau de taille N issu d'un tel sondage, `nb tab a` est le nombre de cases du tableau `tab` qui contiennent `a`.
2) Évaluer la complexité de l'appel de `nb` en fonction de N et de k.
3) En déduire une fonction `elu1` de type `int array -> int -> int` telle que `elu1 tab k` est l'entier donné élu par le tableau `tab` (en supposant que les candidats sont numérotés de 0 à $k-1$) si celui-ci existe et -1 sinon.

4) Évaluer la complexité de votre algorithme en fonction de N et de k.
5) On suppose dans cette question que l'on ne connaît pas le nombre k de candidats. Comment pourrait-on le calculer à partir du seul tableau `tab` ? Modifier votre fonction en conséquence, et donner la nouvelle complexité.

On se propose d'utiliser la stratégie diviser pour régner pour déterminer l'éventuel élu donné par un tableau.

6) Écrire la fonction `miGauche : int array -> int array` qui prend en argument un tableau `tab` de longueur $N > 2$ et retourne le tableau de longueur $\lfloor N/2 \rfloor$ formé par les $\lfloor N/2 \rfloor$ premières cases de `tab`. Quelle est sa complexité ?
7) Écrire la fonction `miDroite : int array -> int array` qui prend en argument un tableau `tab` de longueur $N > 2$ et retourne le tableau de longueur $N - \lfloor N/2 \rfloor$ formé par les $N - \lfloor N/2 \rfloor$ dernières cases de `tab`. Quelle est sa complexité ?
8) Soit `tab` un tableau de longueur $N > 2$. Démontrer que si `tab` donne `a` comme élu alors celui-ci est aussi donné élu par le tableau `miGauche tab` ou par le tableau `miDroite tab`.
9) Proposer une fonction `elu2 : int array -> int * int`, utilisant la stratégie diviser pour régner, telle que si `tab` est un tableau, `elu2 tab` renvoie le couple `(a,n)` lorsque l'entier `a` est donné élu par `tab` et apparait dans n cases exactement de tab, et renvoie le couple `(-1,0)` si `tab` ne donne pas d'élu. On pourra utiliser la fonction `nb` définie en 1) a). On explicitera clairement la stratégie adoptée.
10) On souhaite évaluer la complexité de cette fonction en fonction de N. On suppose dans un premier temps que N est une puissance de 2 : $N = 2^p$, et on note $C(p)$ la complexité de la fonction sur un tableau de taille au plus $N = 2^p$. Donner une relation de récurrence vérifiée par $C(p)$, puis en déduire un ordre de grandeur de $C(p)$ en fonction de p puis de N.
11) Donner la complexité dans le cas général en fonction de N, et comparer avec celle de la fonction `elu1`.