Devoir en temps limité n°5 – 4h

Le code doit être commenté dès qu'il dépasse les 5 lignes ou comprend un concept compliqué. Il est possible (et recommandé) d'utiliser des fonctions auxiliaires en Ocaml, en expliquant leur rôle.

Ce sujet contient 4 exercices indépendants. Des questions de cours sont incluses dans les exercices.

1 Exercice 1 : Un exercice d'induction

Dans cet exercice on étudie un ensemble de points du plan $\mathcal{T}$ défini par induction par :

$\mathcal{B} = \{(3,5)\}$ et $C = \{c_1, c_2, c_3\}$ avec $c_1 : (x,y) \mapsto (x+2, y)$, $c_2 : (x,y) \mapsto (-x, y)$ et $c_3 : (x,y) \mapsto (y,x)$.

Pour clarifier, les éléments de $\mathcal{T}$ sont de la forme (x,y) avec x et y des entiers. Ici, quand on applique un constructeur, on fait le calcul de l'opération "+" ou "-". Par exemple $c_1((3,11)) = (3+2, 11) = (5,11)$.

Des exemple d'éléments de $\mathcal{T}$ sont $(3,5)$ ou $(-3,5) = c_2((3,5))$ ou $(11,5) = c_1^7((3,5))$ (c_1 appliqué 7 fois).

1. Les points suivants sont ils dans $\mathcal{T}$? Si oui, donner comment les obtenir à partir de $(3,5)$. Si non, on ne demande pas de justification.
   - $(8,5)$
   - $(-11,7)$
   - $(3,9)$
2. Donner une méthode pour obtenir tous les points de la forme $(2*n+1, 5)$ pour $n \in \mathbb{N}$ à partir de $(3,5)$.
3. En déduire qu'on peut obtenir tous les points de la forme $(2*n+1, 5)$ pour $n \in \mathbb{Z}$ à partir de $(3,5)$.
4. Montrer finalement qu'on peut obtenir tous les points de la forme $(2*n+1, 2*m+1)$ pour $n, m \in \mathbb{Z}$.

On a montré que tous les points à coordonnées entières impaires sont accessibles. On va maintenant montrer la réciproque.

5. Rappeler le principe de la preuve par induction structurelle. (C'est un théorème du cours qui est attendu).
6. Montrer par induction structurelle que pour tout élément t de $\mathcal{T}$, ses deux coordonées sont impaires.

2 Exercice 2 : Gestion d'une entreprise de livraison

Les algorithmes de cette partie sont à écrire en Ocaml.

Une entreprise de livraison dispose de plusieurs locaux en France et chacun possède plusieurs camions de livraisons. Celle-ci souhaite optimiser le chargement de ses camions pour diminuer ses frais de fonctionnement.

Chaque camion de l'entreprise peut charger une cargaison jusqu'à un poids maximal noté p_{max}. Les clients de l'entreprise lui proposent de livrer des produits numérotés de 1 à n. Pour chaque produit i, l'entreprise dispose de différentes informations provenant de ses clients :

- le poids du produit p_i ;
- la valeur v_i associée au transport du produit : c'est-à-dire l'argent gagné par l'entreprise si elle réalise le transport de ce produit.

L'entreprise cherche donc à trouver un sous-ensemble d'indices notée I contenu dans $\{1, \dots, n\}$ telle que :

$$\sum_{i \in I} p_i \leq p_{max} \text{ (respect du poids maximal)} \tag{1}$$

$$\sum_{i \in I} v_i \text{ soit maximal (optimation du profit pour l'entreprise).} \tag{2}$$

Dans toute la suite, les poids seront donnés en centaines de kilogrammes et les valeurs en centaines d'euros.

Un exemple

Dans cette sous-partie, on suppose que $n = 4$ et que $p_{max} = 8$ centaines de kilogrammes. On stocke alors les différentes informations dans trois tableaux :

- pr est la liste des produits proposés par les clients numérotés de 1 à 4 ;
- p est la liste des poids associés : $p = [3;2;1;4]$;
- v est la liste des valeurs associées : $v = [4;3;1;9]$.

Par exemple, le produit 2 a un poids de deux centaines de kilogrammes et une valeur de trois centaines d'euros.

Les questions 1, 2 et 3 se traitent à l'aide de calculs simples, à faire à la main.

1. Expliquer pourquoi une cargaison constituée d'un, de deux ou de quatre produits ne répond pas au problème posé, c'est-à-dire ne maximise pas le profit fait par l'entreprise en respectant la condition donnée sur le poids maximal.
2. Donner toutes les cargaisons de trois produits respectant le poids maximal. On donnera à chaque fois le profit fait par l'entreprise.
3. Quelle est la cargaison maximisant le profit de l'entreprise ? Que vaut le profit dans ce cas ?

Une méthode intuitive pour la résolution du problème

On garde les notations de la sous-partie précédente dans le cas général :

- Les produits sont numérotés de 1 à n.
- p est la liste des poids associés aux produits : $p = [p_1; \dots; p_n]$;
- v est la liste des valeurs associées aux produits : $v = [v1; \dots; vn]$.

Une méthode intuitive pour tenter d'optimiser le profit de l'entreprise est la suivante : on calcule les ratios $\frac{v_i}{p_i}$, puis on trie les objets par ordre décroissant suivant ces valeurs. Les produits sont alors classés par rentabilité : le premier produit devient le plus rentable " au poids " et ainsi de suite. On ajoute progressivement chaque produit dans la cargaison, dans cet ordre, sans dépasser la limite du poids maximal.

4. Quelle principe de conception d'algorithme cette méthode utilise-t-elle ?
5. Définir une fonction `ratio : float list -> float list -> float list` ayant pour arguments deux listes p et v, où p correspond à la liste des poids et v correspond à la liste des valeurs, renvoyant la liste des ratios $\frac{v_i}{p_i}$.
6. Pour trier les ratios, on va utiliser le tri fusion. Écrire les 3 fonctions nécessaires en Ocaml pour trier **dans l'ordre décroissant** une liste de type `'a list`.
7. Quelle est la complexité de ce tri ? On ne demande pas de justification.

On peut remarquer que si on trie juste les ratios, alors il est difficile de savoir à quel objet chaque ratio est associé. On va donc plutôt trier une liste qui contient les triplets $(\frac{v_i}{p_i}, p_i, v_i)$ pour avoir facilement accès au poids et à la valeur des objets de ratio optimal.

8. Écrire une fonction `zip : float list -> float list -> float*float*float list` qui renvoie la liste des triplets $(\frac{v_i}{p_i}, p_i, v_i)$ à partir des listes p et v.
9. Écrire une fonction `vmax : float list -> float list -> float -> float` prenant en entrée p, v et p_{max} et renvoyant la valeur maximale du profit de l'entreprise en suivant la méthode proposée.
10. On souhaite appliquer cette méthode en utilisant les listes de poids et de valeurs de l'exemple précédent. Donner la liste des ratios, poids et valeurs obtenues à l'aide de la fonction de tri ainsi que le profit obtenu. Commenter le résultat.

Une méthode récursive

On garde les mêmes notations p, v et p_{max} de la partie précédente. Pour simplifier, on considère que toutes les valeurs sont maintenant des **entiers** et que p et v sont des **tableaux**

Nous introduisons une méthode récursive pour résoudre le problème d'optimisation :

- pour chacun des produits, deux choix sont possibles : il fait partie de la cargaison ou non ;
- la récursivité s'effectuera sur les numéros des produits : le premier appel de la fonction se fera en utilisant le produit n, puis le produit $n-1$ et ainsi de suite jusqu'à l'indice 0 (correspondant au cas où il n'y a plus de produits) ;
- Pour $i \in \{0, 1, \dots, n\}$ et $\omega \in 0, 1, \dots, p_{max}$, on note $S(i, \omega)$ la valeur maximale cumulée des produits que l'on peut placer dans un camion d'une capacité maximale (en poids) de ω avec seulement les i premiers produits.

11. Déterminer le ou les cas de base de la fonction $S(i, \omega)$.
12. On va maintenant déterminer la formule récurrente de $S(i, \omega)$ pour des valeurs de i et ω qui ne sont pas des cas de base.
    Indice : il faut séparer le cas où $p_i > \omega$ et le cas où $p_i \leq \omega$. Dans ce deuxième cas, on a le choix de mettre le produit p_i dans le camion ou de ne pas le mettre.
13. Soit $n \in \mathbb{N}^*$ et $p_{max} \in \mathbb{N}^*$. Justifier la terminaison du calcul de $S(n, p_{max})$ par cette formule récurrente.
14. Écrire une fonction récursive `recurrente : int array -> int array -> int -> int -> int` qui prend en entrée p, v, i et ω et calcule $S(i, \omega)$ par la formule déterminée précedemment, **sans aucune amélioration**.

On peut améliorer la complexité temporelle de la méthode récursive par programmation dynamique.

15. Rappeler en une phrase en quoi consiste la programmation dynamique.
16. Écrire une fonction `recurrente_amelioree : int array -> int array -> int -> int -> int` qui prend en entrée p, v, i et ω et calcule $S(i,\omega)$ par **l'approche descendante** de la programmation dynamique.
17. Pour conclure, comment l'entreprise peut-elle utiliser la fonction précédente pour connaitre son gain maximal ?

3 Exercice 3 : Polyômes et méthode de Karatsuba

Les algorithmes de cet exercice doivent être rédigées en C.

Un polynôme $P(X) = \Sigma_{i=0}^{n} a_i X^i$ de degré n est naturellement représenté informatiquement par son tableau de coefficients $[a_0, \dots, a_n]$, donc un tableau **de taille n+1** dont la case i contient le coefficient associé à X^i.

Par exemple, les polynômes $P = X^2 + X + 1$ et $Q = 2X^2 + 1$ peuvent être représentés par les tableaux [1, 1, 1] et [1, 0, 2]. Ils peuvent aussi être représentés par [1, 1, 1, 0, 0, 0] et [1, 0, 2, 0] ou par [1,1,1,0,0] et [1,0,2,0,0,0,0,0].

Dans la suite, pour simplifier, on ne demandera pas d'avoir toujours le tableau le plus court possible et on considèrera qu'un tableau de taille m représente un polynôme de degré $m-1$.

Opérations naïves

La multiplication et l'addition de polynômes se répercutent de façon naturelle sur les tableaux :

$P + Q = 3X^2 + X + 2$ tableau [2,1,3]

$PQ = 2X^4 + 2X^3 + 3X^2 + X + 1$ tableau [1,1,3,2,2]

1. Soit P_1 = [1, 2, 3, 4] et Q_1 = [0, 1, 0, 1]. Calculer $P_1 + Q_1$ et $P_1 Q_1$, sous forme de tableaux.
2. Écrire un algorithme naïf qui additionne ou soustrait deux polynômes. Sa signature sera `int* addition(int* p, int* q, int taillep, int tailleq, bool soustraire)`, qui prend en entrée les tableaux des polynomes P et Q ainsi que leurs tailles.
   Elle prend aussi un paramètre booléen **`soustraire`**. Si celui-ci vaut **`true`**, c'est $P - Q$ qui est calculé. Sinon c'est $P + Q$. Le tableau en sortie devra être de taille $max(taillep, tailleq)$.
3. Écrire un algorithme qui multiplie deux polynômes. Sa signature sera `int* multiplication(int* p, int* q, int taillep, int tailleq)`, qui prend en entrée les tableaux de polynomes P et Q ainsi que leurs tailles. Le tableau en sortie devra être de taille $taillep + tailleq - 1$.
4. Quelle est la complexité des deux algorithmes précédents ?

Une astuce

On va améliorer la complexité de la multiplication de deux polynômes. Une astuce, dite de Karatsuba consiste à remarquer que $ad + bc = ac + bd - (b-a)(d-c)$.

5. En déduire que $(aX + b)(cX + d) = acX^2 + [ac + bd - (b-a)(d-c)]X + bd$.
6. En déduire que si P et Q sont des polynômes de degré 1, on peut trouver le tableau correspondant à PQ en effectuant seulement 3 multiplications et 4 additions.

Pour multiplier deux polynômes de degré n, on peut généraliser l'astuce précédente en une méthode **récursive**.

7. Soit P et Q des polynômes de degré $n = 2^k$, on peut les décomposer en $P = AX^{n/2} + B$ et $P = CX^{n/2} + D$, avec :
   - $deg(A) = deg(C) = n/2$
   - $deg(B) \leq n/2 - 1$ et $deg(D) \leq n/2 - 1$

   Sans justifier, en déduire une formule pour PQ. Combien de multiplications de polynômes de degré inférieur à $n/2$, **et donc d'appels récursifs**, cette formule utilise-t-elle ? Combien d'additions de polynômes utilise-t-elle ?

Programmer la méthode de Karatsuba

On va maintenant programmer la méthode. On reprend les notations de la question précédente.

Pour facilement obtenir les tableaux associés à A, B, C et D, on suppose déjà écrite une fonction `int* extrait(int* p, int deb, int fin)` qui extrait dans un nouveau tableau les éléments de p entre les indices **`deb`** et **`fin`** (inclus).

Par exemple si $P = X^4 + 3X^2 - 2X + 5$, c'est-à-dire $p = [5, -2, 3, 0, 1]$ sous forme de tableau, alors `extrait(p,1,3)` renvoie le tableau $[-2, 3, 0]$, soit le polynôme $3X - 2$.

On suppose également écrite une fonction `int* injecte(int* p, int deb, int fin, int n)` qui renvoie un tableau de taille n rempli de 0, sauf entre les cases deb et fin, où se trouvent les éléments de p.

Par exemple si $P = X + 3$, soit $p = [3, 1]$, alors `injecte(p, 4, 5, 7)` renvoie $[0, 0, 0, 0, 3, 1, 0]$ (soit $X^5 + 3X^4$). **On remarquera qu'injecter à partir de la case deb donne un tableau possible pour le polynôme $P * X^{deb}$.**

Dernière remarque : notre implémentation ne marchera que si n est une puissance de 2, car la formule de la question 7 ne marche que dans ce cas. Mais si on veut multiplier des polynomes d'un degré autre, il suffit de compléter leurs tableaux avec des zéros jusqu'à avoir une bonne taille. Par exemple on peut transformer [1,2,5,6] (polynome de degré 3 et taille 4) en [1,2,5,6,0] (polynome qui a l'air de degré $4 = 2^2$ et taille 5).

8. Compléter le code à trous **en annexe** pour écrire une fonction multipliant deux tableaux selon la méthode de Karatsuba. La fonction agira comme si les entrées de la fonction sont toujours du degré annoncé par leur taille. Le tableau en sortie devra être de taille $2 * taillep - 1$.
   N'oubliez pas de mettre l'annexe dans votre copie.
9. Écrire une formule de récurrence pour la complexité de cet algorithme et la résoudre. On pourra se contenter d'étudier le cas où n est une puissance de 2.

4 Exercice 4 : Arbres croissants

On étudie dans ce problème la structure d'arbre croissant, des arbres binaires étiquetés particuliers.

On rappelle la définition classique des arbres binaires en Ocaml, ici étiquetés par des entiers :

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

Pour t un arbre binaire, on notera $|t|$ son nombre de noeuds et $h(t)$ sa hauteur.

1. Écrire une fonction récursive `taille : arbre -> int` qui calcule le nombre de noeuds d'un arbre et une fonction récursive `hauteur : arbre -> int` qui calcule la hauteur d'un arbre.

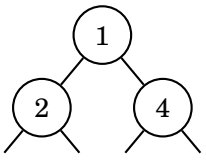
On définit également le nombre d'occurences d'une étiquette y dans un arbre t par induction :

$$\begin{aligned} occ(y, Vide) &= 0 \\ occ(y, N(g, x, d)) &= 1 + occ(y, g) + occ(y, d) \text{ si } y = x \\ occ(y, N(g, x, d)) &= occ(y, g) + occ(y, d) \text{ sinon} \end{aligned}$$

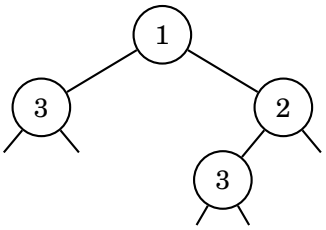
L'ensemble des éléments d'un arbre t est l'ensemble des entiers y pour lesquels $occ(y, t) > 0$.

Structure d'arbre croissant

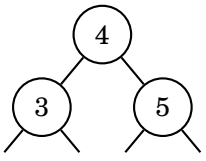
On dit qu'un arbre t est un arbre croissant si, soit $t = Vide$, soit $t = N(g, x, d)$ où g et d sont eux-mêmes deux arbres croissants et x est inférieur ou égal à tous les éléments de g et d. Ainsi les arbres :



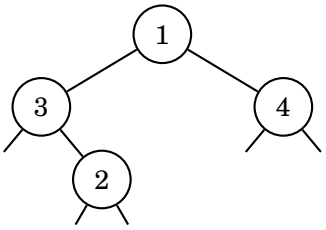
et



sont des arbres croissants mais les arbres



et



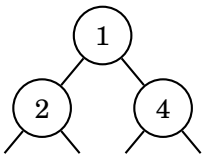
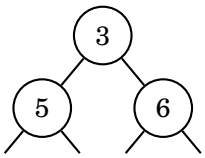
n'en sont pas.

2. Rappeler la définition d'un arbre binaire de recherche. **Comme vous devriez le voir, les arbres croissants et les ABR n'ont rien à voir.**
3. Écrire une fonction `minimum : arbre->int` qui prend en argument un arbre croissant t, et renvoie son plus petit élément. La fonction renverra une erreur si t est vide.
4. Écrire une fonction `est_un_arbre_croissant:arbre->bool` qui prend en argument un arbre t et détermine s'il a la structure d'arbre croissant. On garantira une complexité $O(|t|)$.

Opérations sur les arbres croissants

L'opération de fusion de deux arbres t_1 et t_2 , notée $\texttt{fusion}(t_1, t_2)$, est définie par induction sur la structure des arbres de la manière suivante :

$$\begin{aligned}
\texttt{fusion}(t_1, Vide) &= t_1 \\
\texttt{fusion}(Vide, t_2) &= t_2 \\
\texttt{fusion}(N(g_1, x_1, d_1), N(g_2, x_2, d_2)) &= N(\texttt{fusion}(d_1, N(g_2, x_2, d_2)), x_1, g_1) \text{ si } x_1 \leq x_2 \\
\texttt{fusion}(N(g_1, x_1, d_1), N(g_2, x_2, d_2)) &= N(\texttt{fusion}(d_2, N(g_1, x_1, d_1)), x_2, g_2) \text{ sinon}
\end{aligned}$$

5. Donner le résultat de la fusion des arbres croissants  et 

6. Soit t le résultat de la fusion de deux arbres croissants t_1 et t_2 . Montrer que t possède la structure d'arbre croissant et que, pour tout entier x, $occ(x, t) = occ(x, t_1) + occ(x, t_2)$
7. Déduire de l'opération **fusion** une fonction `ajoute:int->arbre->arbre` qui prend en arguments un entier x et un arbre croissant t et renvoie un arbre croissant t' tel que $occ(x, t') = 1 + occ(x, t)$ et $occ(y, t') = occ(y, t)$ pour tout $y \neq x$.
8. Déduire de l'opération **fusion** une fonction `supprime_minimum : arbre->arbre` qui prend en argument un arbre croissant t, en supposant $t \neq Vide$, et renvoie un arbre croissant t' tel que, si m désigne le plus petit élément de t, on a $occ(m, t') = occ(m, t) - 1$ et $occ(y, t') = occ(y, t)$ pour tout $y \neq m$.
9. Soient $x_0, \ldots, x_{n-1}$ des entiers et $t_0, \ldots, t_n$ les $n + 1$ arbres croissants définis par $t_0 = Vide$ et $t_{i+1} = \texttt{fusion}(t_i, N(Vide, x_i, Vide))$ pour $0 \leq i < n$. Écrire une fonction `ajouts_successifs : int array -> arbre` qui prend en argument un tableau contenant les entiers $x_0, \ldots, x_{n-1}$ et qui renvoie l'arbre croissant t_n.
10. Avec les notations de la question précédente, donner, pour tout n, des valeurs $x_0, \ldots, x_{n-1}$ qui conduisent à un arbre croissant t_n de hauteur au moins égale à $n/2$.

5 Annexe

```
int* karatsuba(int* p, int* q, int taillep){
   /*Entrées : le tableau p,
               le tableau q,
               leur taille commune taillep.*/

   if (                        ) { //Cas de base




   }
   else{
       //Attention, le degré n'est pas la taille.
       int n = taillep-1;

       //Définir A,B,C et D.
       //On les crée tous de taille n/2+1 (donc de degré n/2)
       int* A = extrait(
       int* B = extrait(
       B[  ] = 0;
       int* C = extrait(
       int* D = extrait(
       D[  ] = 0;

       //Définir AC,BD, B-A, D-C, (B-A)(D-C) et la somme de tous
       int* AC =
       int* BD =
       int* BmoinsA = addition(
       int* DmoinsC = addition(
       int* BmoinsADmoinsC =
       int* somme = addition(addition(

       //Utiliser la formule
       int* ACXn = injecte(AC,
       int* sommeXnsur2 = injecte(somme,
       int* res =

       //Que faut-il ne pas oublier de faire avec ses tableaux en C ?




       //Finalisation
       return res;}}
```