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

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.*
   - $(8,5)$ pas un élément de $\mathcal{T}$.
   - $(-11,7) = c_3(c_1(c_3(c_2(c_1^4((3,5))))))$.
   - $(3,9) = c_3(c_1^2(c_3((3,5))))$.
2. *Donner une méthode pour obtenir tous les points de la forme $(2*n+1,5)$ pour $n \in \mathbb{N}$ à partir de $(3,5)$.*
   Soit $n \in \mathbb{N}^*$. On a $2n+1-3 = 2n-2$, donc $c_1^{n-1}((3,5)) = (3+2*(n-1),5) = (2n+1,5)$.
   La méthode est donc d'appliquer $n-1$ fois c_1 sur $(3,5)$.
   Pour $n=0$, on transforme 3 en -3 avec c_2, on ajoute 2 avec c_1, on obtient -1 et on réapplique c_2 pour avoir 1.
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 $(3,5)$.*
   Si $n \geq 0$, on utilise la méthode précédente.
   Si $n < 0$, alors $2n+1 < 0$. Donc $-(2n+1)$ est de la forme $2m+1$ avec $m \in \mathbb{N}$. On applique la méthode précédente pour obtenir $(2m+1,5)$, puis on applique c_2 : $c_2((2m+1,5)) = c_2((-(2m+1),5)) = c_2((2n+1,5))$.
4. *Montrer finalement qu'on peut obtenir tous les points de la forme $(2*n+1,2*m+1)$ pour $n,m \in \mathbb{Z}$.*
   En utilisant la question précédente, on obtient $(2n+1,5)$ à partir de $(3,5)$.
   Ensuite on utilise c_3 pour obtenir $(5,2n+1)$. On remarque que $3+2$, donc en appliquant la méthode précédente en retirant une application de c_1, on obtient $(2m+1,2n+1)$. En réutilisant c_3, on obtient $(2n+1,2m+1)$.
5. *Rappeler le principe de la preuve par induction structurelle.*
   Soit P la propriété à montrer. Si on peut montrer :
   - $\forall b \in \mathcal{B}$, $P(b)$ est vrai.
   - $\forall c \in C$ d'arité k, $\forall t_1,\dots,t_k \in \mathcal{T}$, $P(c(t_1,\dots,t_k))$ est vrai.

   alors $\forall t \in \mathcal{T}$, $P(t)$ est vrai.
6. *Montrer par induction structurelle que pour tout élément t de $\mathcal{T}$, ses deux coordonées sont impaires.*
   On raisonne par induction structurelle sur la structure de t.
   - Si $t \in \mathcal{B}$, alors $t = (3,5)$ qui a deux coordonnées impaires.
   - Si $t = c_1(t')$, en supposant $P(t')$, c'est à dire qu'il existe n,m des entiers impairs tels que $t' = (2n+1,2m+1)$, alors $t = (2n+3,2m+1)$, donc $P(t)$.
   - Si $t = c_2(t')$, en supposant $P(t')$, c'est à dire qu'il existe n,m des entiers tels que $t' = (2n+1,2m+1)$, alors $t = (-2n-1,2m+1)$, donc $P(t)$.
   - Si $t = c_3(t')$, en supposant $P(t')$, c'est à dire qu'il existe n,m des entiers tels que $t' = (2n+1,2m+1)$, alors $t = (2m+1,2n+1)$, donc $P(t)$.

   Par principe d'induction, on peut affirmer que tout éléments de $\mathcal{T}$ a deux coordonnées impaires.

2 Exercice 2 : Gestion d'une entreprise de livraison

Un exemple

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.*
   Si on met un objet ou deux objets quelconques dans le camion, la quantité de poids restante permet toujours de rajouter un autre objet.
   Mettre les 4 objets dépasse le poids max : $1+2+3+4 = 10 > 8$.

2. *Donner toutes les cargaisons de trois produits respectant le poids maximal. On donnera à chaque fois le profit fait par l'entreprise.*
   Produits 1 et 2 et 3 : valeur 8.
   Produits 1 et 3 et 4 : valeur 14.
   Produits 2 et 3 et 4 : valeur 13.
3. *Quelle est la cargaison maximisant le profit de l'entreprise ? Que vaut le profit dans ce cas ?*
   Il faut mettre les objets 1 et 3 et 4 dans le camion. Le profit est de 1400 euros.

Une méthode intuitive pour la résolution du problème

4. *Quelle méthode de conception d'algorithme cette méthode utilise-t-elle ?*
   C'est un algorithme glouton. À chaque étape, il choisit d'ajouter l'objet optimal en termes de ratio.
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* $\frac{v_i}{p_i}$.

```
let rec ratio p v = match p,v with
  |[],[] -> []
  |tp::qp, tv::qv -> tv/.tp::(ratio qp qv)
  |_-> failwith "listes de tailles différentes";;
```

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`.
   Attention on veut l'ordre décroissant !

```
let rec separe l = match l with
  |[] -> [],[]
  |[e] -> [e],[]
  |t1::t2::q -> let l1,l2 = separe q in t1::l1,t2::l2;;

let rec fusion l1 l2 = match l1,l2 with
  |[],_ -> l2
  |_,[] -> l1
  |t1::q1, t2::q2 -> if t1>t2 then t1::(fusion q1 l2) else t2::(fusion l1 q2);;

let rec tri_fusion l = match l with
  |[] -> []
  |[e] -> [e]
  |t::q -> let l1,l2 = separe l in fusion (tri_fusion l1) (tri_fusion l2);;
```

7. *Quelle est la complexité de ce tri ? On ne demande pas de justification.*
   C'est $O(n\log_2(n))$ où n est la taille de la liste à trier.
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.*

```
let zip p v = match p,v
  |[],[] -> []
  |tp::qp, tv::qv -> (tv/.tp,tp,tv)::(ratio qp qv)
  |_-> failwith "listes de tailles différentes";;
```

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.*

```
let vmax p v pmax =
  let ratios = tri_fusion (zip p v) in

  let rec construit_chargement lr prest =
      (*Prend en entrée lr la liste avec les ratios et prest le poids qui reste.
      Renvoie la valeur de la cargaison optimale selon le glouton*)
  match r with
  |[] -> 0
  |(r,p,v)::q -> if prest+p<=pmax then (*Il reste assez de place pour l'objet optimal*)
                  v + construit_chargement q (prest-p)
                 else construit_chargement q prest (*L'objet optimal est trop gros, on tente le suivant*)
  in construit_chargement ratios pmax;;
```

10. *On souhaite appliquer cette méthode en utilisant les listes de poids et de valeurs de l'exemple précédent. Donner la liste des ratios, les listes de poids et de valeurs obtenues à l'aide de la fonction de tri ainsi que le profit obtenu. Commenter le résultat.*
    Les ratios sont [1,33 ;1,5 ;1 ;2,25].
    L'objet 4 est choisi. Il reste 4 centaines de kg.
    L'objet 2 est choisi. Il reste 2 centaines de kg.
    L'objet 1 est testé mais il est trop gros.
    L'objet 3 est choisi.
    La solution proposée par le glouton est de valeur 13, non optimale.

Une méthode récursive

11. *Déterminer le ou les cas de base de la fonction $S(i,\omega)$.*
    $S(0,\omega)=0$ (et $S(i,0)=0$, mais cette condition ne sert pas vraiment en pratique)
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.*
    - Si $p_i > \omega$ alors l'objet i est inutilisable. Donc la meilleure valeur qu'on puisse obtenir est la même que si on avait que les objets jusqu'au rang $i-1$.
      D'où $S(i,\omega)=S(i-1,\omega)$.
    - Sinon
      - On peut choisir de mettre p_i dans le camion. On se ramène donc à remplir un camion de taille $\omega - p_i$ de manière optimale avec les objets restants (ceux d'indice inférieur à $i-1$).
        Dans ce cas le mieux qu'on puisse faire est $v_i + S(i-1,\omega-p_i)$
      - On peut choisir de ne pas mettre p_i dans le camion.
        Dans ce cas le mieux qu'on puisse faire est $S(i-1,\omega)$

      Comme on ne sait pas lequel des deux choix est le meilleur, on affirme juste que $S(i,\omega)=\max(S(i-1,\omega-p_i)+v_i, S(i-1,\omega))$.
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.*
    i est un variant des appels récursifs, puisque tous les appels se font avec $i-1<i$ et la récursion s'arête si i atteint 0.
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**.*

```
let rec recurrente p v i w = match i,w with
  |0,_ -> 0
  |_,0 -> 0
  |_,_ when p.(i-1)>w -> recurrente p v (i-1) w
  |_,_ -> max (recurrente p v (i-1) w) ((recurrente p v (i-1) (w-p.(i-1))) +v.(i-1));;
```

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

15. *Rappeler en une phrase en quoi consiste la programmation dynamique.*
    On va garder en mémoire les calculs fait au fur et à mesure des appels récursifs pour éviter de calculer deux fois une valeur $S(i,\omega)$ avec les mêmes i et ω.
16. *Écrire une fonction `recurrente_amelioree : int list -> int list -> 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.*

```
let recurrente_amelioree p v i w =
  (*On crée une matrice pour mémoiser. Initialement, elle est remplie de -1*)
  let res = Array.make_matrix (i+1) (w+1) -1 in

  let rec aux i w =
      if mat.(i).(w) = -1 then (*Si la case est vide, on remplit la case*)
      begin
          match i,w with
          |0,_ -> mat.(i-1).(w)<-0
          |_,0 -> mat.(i-1).(w)<-0
          |_,_ when p.(i-1)>w -> mat.(i-1).(w) <- aux (i-1) w
          |_,_ -> mat.(i-1).(w) <- max (aux (i-1) w) ((aux (i-1) (w-p.(i-1))) +v.(i-1))
      end;
      mat.(i-1).(j) (*La case n'est pas/plus vide, on renvoie son contenu*)
  in aux i w;;
```

17. *Pour conclure, comment l'entreprise peut-elle utiliser la fonction précédente pour connaitre son gain maximal ?*
    Il lui suffit de calculer `recurrente_amelioree p v (Array.length p) pmax`.

3 Exercice 3 : Polyômes et méthode de Karatsuba

Opérations naïves

1. *Soit P_1 = [1, 2, 3, 4] et Q_1 = [0, 1, 0, 1]. Calculer $P_1 + Q_1$ et P_1Q_1, sous forme de tableaux.*
   $P_1 + Q_1$ peut être représenté par [1,3,3,5].
   P_1Q_1 peut être représenté par [0,1,2,4,6,3,4].
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 $max(taillep, tailleq)$.*

```c
int* addition(int* p, int* q, int taillep, int tailleq, bool soustraire){
  //Créer le tableau résultat
  int* res;
  int tailleres,mini;
  if(taillep>=tailleq){ //La taille du résultat est max(taillep, tailleq)
      res = malloc(taillep*sizeof(int));

      //Appliquer la formule pour l'addition
      for(int i=0; i<tailleq; i+=1){
          if(soustraire == true){res[i] = p[i]-q[i];}
          else{res[i] = p[i]+q[i];}
      }

      for(int i=tailleq;i<taillep;i+=1){ //Le degré de p est plus grand,
      //on recopie les coefficients restants.
          res[i] = p[i];
      }

      return res;
  }
  else{
      res = malloc(tailleq*sizeof(int));

      //Appliquer la formule pour l'addition
      for(int i=0; i<taillep; i+=1){
          if(soustraire == true){res[i] = p[i]-q[i];}
          else{res[i] = p[i]+q[i];}
      }

      for(int i=taillep;i<tailleq;i+=1){ //Le degré de q est plus grand,
      //on recopie les coefficients restants (avec un - si nécessaire)
          if(soustraire == true){res[i] = -q[i];}
          else{res[i] = q[i];}
      }

      return res;
  }
}
```

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 $taillep + tailleq - 1$.*

```c
int* multiplication(int* p, int* q, int taillep, int tailleq){
  //Créer le tableau résultat
  int* res = malloc((taillep+tailleq-1)*sizeof(int));
  for(int i=0;i<taillep+tailleq-1;i+=1){
      res[i]=0;
  }

  //Appliquer la formule pour la multiplication
  for(int i=0; i<taillep; i+=1){
      for(int k=0; k<tailleq; k+=1){
          res[i+k] += p[i]*q[k];
      }
  }
```

```
    return res;
}
```

4. Quelle est la complexité des deux algorithmes précédents ?
   L'algorithme d'addition contient deux boucles for effectuant au total $\max(taillep, tailleq)$ itérations. Chaque itération a un cout constant. Le cout des boucles est donc linéaire.
   L'algorithme d'addition a donc une complexité linéaire en $O(max(taillep, tailleq))$.
   L'algorithme de multiplication contient lui deux boucles imbriquées. La seconde fait $tailleq$ itérations. Chaque itération coute 3 opérations arithmétiques. La première fait $taillep$ itérations, chaque itération coutant donc $3tailleq$ opérations.
   Au total le cout est un $O(taillep * tailleq)$.

Une astuce

5. *En déduire que* $(aX+b)(cX+d) = acX^2 + [ac+bd-(b-a)(d-c)]X + bd$.
   $(aX+b)(cX+d) = acX^2 + [ad+bc]X + bd = acX^2 + [ac+bd-(b-a)(d-c)]X + bd$
6. *En déduire que que l'on peut multiplier deux polynômes de degré 1 en effectuant seulement 3 multiplications et 4 additions.*
   Pour calculer le tableau correspondant à $(aX+b)(cX+d)$, il suffit de calculer ac (1 multiplication), bd (1 multiplication), $b-a$ (1 addition), $d-c$ (1 addition), $(b-a)(d-c)$ (1 multiplication puisqu'on a déjà calculé les facteurs) et $ac+bd-(b-a)(d-c)$ (2 additions puisqu'on a déjà calculé chaque terme). Au total cela fait 3 multiplications et 4 additions d'entiers.
   Ensuite il suffit de mettre ab dans la case 2, $ac+bd-(b-a)(d-c)$ dans la case 1 et bd dans la case 0.

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 $P = CX^{n/2} + D$*
   Sans justifier, en déduire une formule pour PQ. Combien de multiplications de polynômes de degré inférieur à $n/2$ cette formule utilise-t-elle ? Combien d'additions de polynômes utilise-t-elle ?
   En appliquant le même principe que dans les questions précédentes, on a $(AX^{n/2}+B)(CX^{n/2}+D) = ACX^n + [AC+BD-(B-A)(D-C)]X^{n/2} + BD$.
   Cette formule utilise 3 multiplications de polynomes de degré $n/2$ et 4 additions de polynomes de degré $n/2$.
   Ici il nous faut 3 additions en plus car les polynomes ACX^n, $[AC+BD-(B-A)(D-C)]X^{n/2}$ et BD ont des monomes X^i en commun. Le calcul de ACX^n ou $[AC+BD-(B-A)(D-C)]X^{n/2}$ ne nécessite par ailleurs pas de multiplication, comme on le verra dans la suite.

Programmer la méthode de Karatsuba

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 $2 * taillep - 1$.*
   On utilise `extrait` pour récupérer les tableaux des polynomes A, B, C et D. Pour simplifier la manipulation des degré, ils sont tous de taille $n/2+1$.
   Les produits AC, BD et $(B-A)(D-C)$ sont calculés récursivement par la méthode de Karatsuba.
   Les sommes $B-A$, $D-C$ et $AC+BD-(B-A)(D-C)$ sotn calculées avec la fonction `addition` (qui est optimale en temps).
   Ensuite on calcule ACX^n et $[AC+BD-(B-A)(D-C)]X^{n/2}$ avec la fonction `injecte`. Il suffit de décaler les coefficients de AC de n zéros vers la droite et les coefficients de $AC+BD-(B-A)(D-C)$ de $n/2$ zéros vers la droite.
   Enfin, on additionne ACX^n, $[AC+BD-(B-A)(D-C)]X^{n/2}$ et BD avec la fonction addition.
   La difficulté principale, outre comprendre le principe, est de choisir correctement les indices et les tailles des tableaux.

```
int* karatsuba(int* p, int* q, int taillep){
   /*Entrées : le tableau p,
               le tableau q,
               leur taille commune n*/
   if (taillep == 2) { /*On peut prendre le degré 1 comme cas de base et utiliser
   la formule en q5, ou juste la formule normale*/
       int* res = malloc(3*sizeof(int));
```

```
        res[0]=p[0]*q[0];
        res[2]=p[1]*q[1];
        res[1]=p[1]*q[0]+q[1]*p[0];
        return res;
    }
    else{
        //Attention, la taille n'est pas le degré.
        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(p,n/2,n);
        int* B = extrait(p,0,n/2);
        B[n/2] = 0;
        int* C = extrait(q,n/2,n);
        int* D = extrait(q,0,n/2);
        D[n/2] = 0;


        //Définir AC,BD, B-A, D-C, (B-A)(D-C) et la somme de tous
        int* AC = karatsuba(A,C,n/2+1);
        int* BD = karatsuba(B,D,n/2+1);
        int* BmoinsA = addition(B,A,n/2+1,n/2+1,true);
        int* DmoinsC = addition(D,C,n/2+1,n/2+1,true);
        int* BmoinsADmoinsC =karatsuba(BmoinsA, DmoinsC, n/2+1);
        int* somme = addition(addition(AC,BD,n+1,n+1,false), BmoinsADmoinsC, n+1, n+1, true);

        //Utiliser la formule
        int* ACXn = injecte(AC, n, 2*n, 2*n+1);
        int* sommeXnsur2 = injecte(somme, n/2, n+n/2, 2*n+1);
        int* res = addition(addition(ACXn, sommeXnsur2, 2*n+1, 2*n+1, false), BD, 2*n+1, n+1, false);

        //Que faut-il ne pas oublier de faire avec ses tableaux en C ?
        free(A);free(B);free(C);
        free(AC); free(BD); free(BmoinsA); free(DmoinsC); free(BmoinsADmoinsC); free(somme);
        free(ACXn); free(sommeXnsur2);
        //Comme on me l'a fait remarquer, il y a quelques tableaux créés qu'on ne peut pas supprimer
        //(quand on applique addition sur addition).
        //C'est une erreur de design qu'il faudrait corriger.

        //Finalisation
        return res;
    }
}
```

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.*
   Soit $n = 2^k$.
   La formule de complexité de cet algorithme est

$$C(1) = 7$$
$$C(n) = 3 * C(n/2) + A * n$$

   où A est une certaine constante, liée au fait que la fonction d'addition écrite plus haut a une complexité en $O(n)$.

   Montrons par récurrence sur l que $\forall l \in [|0, k|], C(n) = 3^l C(n/2^l) + \sum_{i=0}^{l-1} An/2^i * 3^i$

   - Pour $l = 0$, $C(n) = 3^0 C(n/2^0) + 0$ est vrai.
   - Supposons $C(n) = 3^l C(n/2^l) + \sum_{i=0}^{l-1} An/2^i * 3^i$. En utilisant la formule de récurrence sur $C(n/2^l)$, on obtient $C(n/2^l) = 3C(n/2^{l+1}) + An/2^l$.
     Donc $C(n) = 3^{l+1} C(n/2^{l+1}) + An/2^l * 3^l + \sum_{i=0}^{l-1} An/2^i * 3^i = 3^{l+1} C(n/2^{l+1}) + \sum_{i=0}^{l} An/2^i * 3^i$

   La formule est correcte par principe de récurrence. En particulier elle est vraie en $l = k$, donc

$$
\begin{aligned}
C(n) &= 3^k C(1) + \sum_{i=0}^{k-1} An/2^i * 3^i \\
&= 7 * 3^k + 2An((3/2)^k - 1) \text{ en reconnaissant la somme d'une suite géométrique} \\
&= 3^{\log_2(n)} + 2An(3/2)^{\log_2(n)} - 2An \\
&= n^{\log_2(3)} + 2An^{1+\log_2(3/2)} - 2An \text{ en utilisant la définition de la puissance généralisée} \\
&= (2A+1)n^{\log_2(3)} - 2An \text{ en remarquant que} 1 + \log_2(3/2) = log_2(3)
\end{aligned}
$$

Au final $C(n) = O(n^{\log_2(3)}) = O(n^{1,6})$, ce qui est légérement mieux que le $O(n^2)$ de la version naïve !

4 Exercice 4 : Arbres croissants

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

```
let rec taille a = match a with
   |Vide -> 0
   |N(g,x,d) -> 1+ (taille g) + (taille d);;

 let rec hauteur a = match a with
   |Vide -> -1
   |N(g,x,d) -> 1+max (hauteur g) (hauteur d);;
```

Structure d'arbre croissant

2. *Rappeler la définition d'un arbre binaire de recherche.*
   Un ABR est un arbre dans lequel tout noeud a une étiquette supérieure aux étiquettes apparaissant dans son sous-arbre gauche et inférieure à celles apparaissant dans son sous-arbre droit.
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.*
   On remarque que le minimum d'un arbre croissant est toujours sa racine. (en cas de multiples occurences du minimum, une des occurences est à la racine)

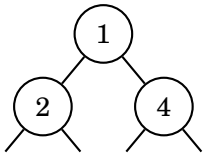
```
let minimum a = match a with
  |Vide -> failwith "arbre vide"
  |N(g,x,d) -> x;;
```

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* $O(|t|)$.
   Il suffit de tester que tout noeud de l'arbre a une étiquette inférieure à celle de son fils gauche et de son fils droit.

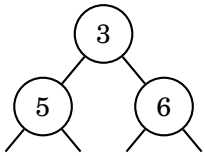
```
let rec est_un_arbre_croissant a = match a with
  |Vide -> true
  |N(g,x,d) -> match g,d with
                |Vide,Vide -> true
                |N(gg,eg,gd),Vide -> (x<=eg) && (est_un_arbre_croissant g)
                |Vide, N(dg,ed,dd) -> (x<=ed) && (est_un_arbre_croissant d)
                |N(gg,eg,gd), N(dg,ed,dd) -> (x<=ed) && (x<=eg) && (est_un_arbre_croissant d)
                                             && (est_un_arbre_croissant g);;
```

Opérations sur les arbres croissants

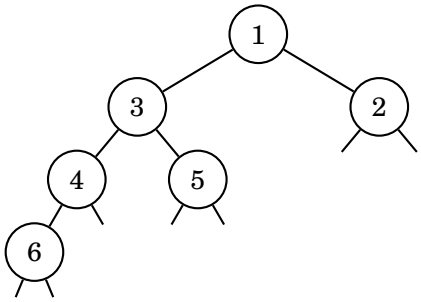
5. *Donner le résultat de la fusion des arbres croissants*



et



Le résultat est l'arbre suivant :



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, $occ(x,t) = occ(x,t_1) + occ(x,t_2)$.*
   On le montre par induction sur les appels de la fonction **fusion**.
   - Si $t_2 = Vide$, alors $t =$**fusion**$(t_1,t_2) = t_1$ est un arbre croissant. De plus pour tout entier $occ(x,t) = occ(x,t_1) = occ(x,t_1) + occ(x,t_2)$.
   - Si $t_1 = Vide$ on conclut par le même raisonnement.
   - Si $t_1 = N(g_1,x_1,d_1)$ et $t_2 = N(g_2,x_2,d_2)$ avec $x_1 \leq x_2$, alors $t = N($**fusion**$(d_1,N(g_2,x_2,d_2)),x_1,g_1)$. Supposons par hypothèse d'induction que **fusion**$(d_1,N(g_2,x_2,d_2))$ renvoie un arbre t' qui est croissant et tel que pour tout entier x, $occ(x,t') = occ(x,d_1) + occ(x,t_2)$.
     Alors $t = N(t',x_1,g_1)$ est croissant car
     - Toute étiquette de g_1 est plus grande que x_1 puisque t_1 est croissant.
     - Les étiquettes de t' sont soit des étiquettes de d_1, donc plus grande que x_1, soit des étiquettes de t_2. Or toute étiquette x de t_2 vérifie $x \geq x_2 \geq x_1$.

     De plus $occ(x,t) = occ(x,t') + occ(x,g_1) + 1$ si $x = x_1$ et $occ(x,t) = occ(x,t') + occ(x,g_1)$ sinon.
     On peut réécrire ceci en $occ(x,t) = occ(x,d_1) + occ(x,t_2) + occ(x,g_1) + 1 = occ(x,t_1) + occ(x,t_2)$ dans un cas et $occ(x,t) = occ(x,d_1) + occ(x,t_2) + occ(x,g_1) = occ(x,t_1) + occ(x,t_2)$ dans l'autre.
   - Si $x_2 \leq x_1$, on raisonne de la même manière.

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 $y \neq x$.*
   Pour ajouter, il suffit de fusionner t avec l'arbre $N(Vide,x,Vide)$. On réécrit la récurrence de **fusion** quand $d_2 = Vide$ et $g_2 = Vide$.

```
let rec ajoute x t = match t with
   |Vide -> N(Vide,x,Vide)
   |N(g,r,d) when r>=x -> N(t,x,Vide)
   |N(g,r,d) when r<=x -> N(ajoute x d, r, g)
```

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 $y \neq m$.*
   Si t est de la forme $N(g,m,d)$ où m est forcément le minimum, alors pour garder les autres étiquettes dans un abre croissant, il suffit de fusionner g et d.

```
let supprime_minimum t =
   let rec fusion t1 t2 = match t1,t2 with
       |Vide,_ -> t2
       |_,Vide -> t1
       |N(g1,x1,d1),N(g2,x2,d2) when x1<=x2 -> N(fusion d1 t2, x1,g1)
       |N(g1,x1,d1),N(g2,x2,d2) -> N(fusion d2 t1, x2, g2)

match t with
   |Vide -> failwith "arbre vide"
   |N(g,m,d) -> fusion g d;;
```

9. *Soient $x_0,\dots,x_{n-1}$ des entiers et $t_0,\dots,t_n$ les $n+1$ arbres croissants définis par $t_0 = Vide$ et $t_{i+1} = fusion(t_i,N(Vide,x_i,Vide))$ pour $0 \leq i < n$. Écrire une fonction `ajouts_successifs : int array -> arbre` qui prend en argument un tableau contenant les entiers $x_0,\dots,x_{n-1}$ et qui renvoie l'arbre croissant t_n.*

```
let ajouts_successifs tab =
   let rec aux n = match n with
   |0 -> Vide
   |_ -> let t = aux (n-1) in ajoute tab.(n-1) t in
   aux (Array.length tab);;
```

10. *Avec les notations de la question précédente, donner, pour tout n, des valeurs $x_0, \dots, x_{n-1}$ qui conduisent à un arbre croissant t_n de hauteur au moins égale à $n/2$.*

    Si on considère les valeurs $n-1, \dots, 0$ dans cet ordre, c'est à dire $x_i = n-1-i$, alors l'arbre construit est de hauteur peigne $n-1$.

    Montrons le par récurrence sur n. Pour $n = 1$, l'arbre est $N(Vide, x_0, Vide)$ est de hauteur 0.

    Si t_n est un peigne de hauteur $n-1$, alors $t_{n+1} = N(\texttt{fusion}(Vide, t_n), x_{n+1}, Vide) = N(t_n, x_{n+1}, Vide)$ est un peigne de hauteur n.