Mpsi 1 et 2 - option informatique **Devoir surveillé 4**

Les algorithmes seront écrits en code CAML et seront accompagnés de commentaires et d'explications qui permettront de les comprendre aisément.

Le sujet est constitué de deux problèmes indépendants, l'un sur les tas, l'autre sur de la programmation dynamique.

Problème 1 - Tas

Dans un arbre binaire non vide, on appelle hauteur d'une feuille le nombre d'arêtes du chemin la reliant à la racine. La hauteur de l'arbre lui-même est alors le maximum des hauteurs de ses feuilles. Par convention, on décrète que l'arbre vide est de hauteur -1.

Dans ce problème, on considère des arbres binaires d'entiers, auxquels on ajoute une information booléenne en chaque sommet, représentés en CAML par le type suivant :

```
type arbre = V | S of int * bool * arbre * arbre
```

Le symbole V représente l'arbre vide et le symbole S(x, b, fg, fd) représente l'arbre dont la racine a pour valeur l'entier x, de fils gauche fg et de fils droit fd, le symbole b désignant un booléen auquel on donnera une signification plus tard.

On définit la fonction valeur

```
let valeur t =
  match t with
  | V -> failwith "arbre vide"
  | S(x, _, _, _) -> x;;
```

Partie 1 - Généralités

Question 1) Écrivez une fonction hauteur t (t comme « tree ») de type `arbre -> int`, qui calcule la hauteur d'un arbre binaire.

Question 2) Montrez par récurrence sur h qu'un arbre binaire de hauteur h a au maximum 2^h feuilles.

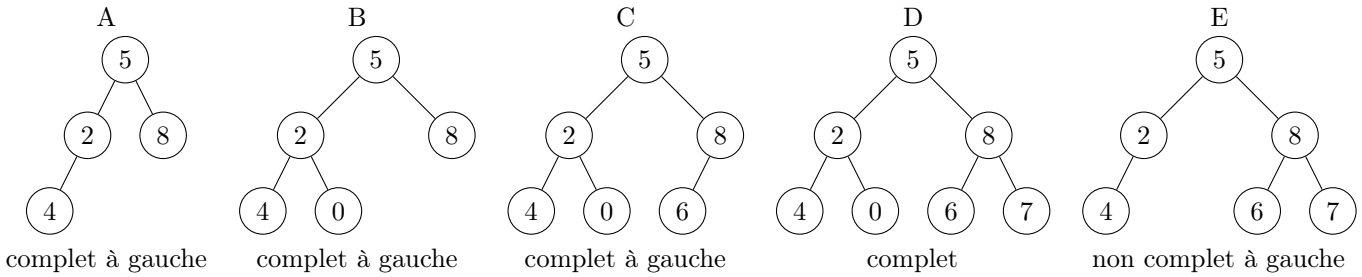
Partie 2 - Arbres complets à gauche

Un arbre binaire de hauteur $h \geqslant 0$ est dit complet à gauche (en abrégé ACG) quand tous les niveaux de l'arbre de la racine jusqu'à la hauteur $h-1$ sont remplis (autrement dit tous les sommets de hauteur $k \leqslant h-2$ ont deux fils), toutes ses feuilles sont de hauteur h ou $h-1$ et dans sa représentation graphique habituelle, elles sont poussées vers la gauche (autrement dit, celles de hauteur h sont à gauche sur la dernière ligne de l'arbre).

Un arbre binaire de hauteur $h \geqslant 0$ est dit complet (tout court) quand il a exactement 2^h feuilles. En particulier, un arbre qui est réduit à sa racine (donc de hauteur 0) est complet. Par convention, l'arbre vide est complet.

Avec ces définitions, il est alors évident qu'un arbre complet est aussi complet à gauche.

Voyons quelques exemples :



On constate qu'on peut diviser les ACG en 4 catégories, comme dans les exemples ci-dessus.

- A : Le fils gauche de la racine n'est pas complet, le fils droit est complet, ce qui implique que le fils gauche est plus haut que le droit ;
- B : Les deux fils de la racine sont complets, mais le fils gauche est plus haut que le fils droit ;
- C : Le fils gauche est complet, le fils droit ne l'est pas, ce qui implique qu'ils sont de même hauteur ;
- D : L'arbre est complet, ses deux fils le sont aussi et sont de même hauteur.

Précisons maintenant la valeur du booléen dans la définition de l'arbre : il indique si l'arbre est complet ou non. Plus précisément, en chaque sommet s de l'arbre, le booléen indique si le sous-arbre de racine s est complet ou pas.

On définit donc la fonction complet :

```
let complet t =
  match t with
  | V -> true
  | S(_, b, _, _) -> b;;
```

Dans toute la suite du problème, **tous les arbres sont supposés être complets à gauche** et si des fonctions calculent des arbres, vous ferez attention que ceux-ci le soient aussi.

Question 1) On considère un ACG non vide `t = S(x, b, fg, fd)`. On appelle signature de l'arbre le triplet `(b, bg, bd)` où `b = complet(t)`, `bg = complet(fg)` et `bd = complet(fd)`.

Donnez les valeurs possibles de la signature de l'arbre `t` et justifiez que la connaissance de cette signature permet de savoir dans quelle catégorie est l'arbre `t` : vous n'hésiterez pas à illustrer votre propos de schémas représentant des arbres.

Question 2) Quand un ACG de hauteur h est non vide, on appelle dernier élément de l'arbre l'entier de sa feuille la plus à droite parmi les feuilles de hauteur h.

a) Justifiez l'appellation « dernier » en évoquant un parcours de l'arbre.

b) Sur les 4 exemples précédents, donnez le dernier élément de chaque ACG.

c) Selon la signature de l'arbre, expliquez comment retrouver récursivement le dernier élément de l'arbre.

d) Écrivez une fonction `dernier t` de type `arbre -> int`, qui calcule le dernier élément de l'ACG `t`. Dans le cas de l'arbre vide, on interrompt le calcul par un `failwith "message"`.

e) Quelle est la complexité de votre fonction, en fonction de h, hauteur de l'arbre ?

Question 3) Les exemples montrent comment on a obtenu successivement les ACG : on obtient l'arbre B en ajoutant l'entier 0 à l'arbre A, on obtient l'arbre C en ajoutant l'entier 6 à l'arbre B et pareil pour D à partir de C. À chaque fois, on place le nouvel élément « au bout » de l'arbre.

a) Selon la signature de l'arbre, expliquez où ajouter récursivement l'élément voulu dans l'arbre.

b) Dans chacun des cas précédents, quelle est le booléen à placer dans la racine du nouvel arbre ?

c) Écrivez une fonction `ajouter x t` de type `int -> arbre -> arbre`, qui prend en paramètres un entier et un ACG et qui calcule un ACG obtenu en ajoutant une nouvelle feuille au bout de l'arbre.

d) Quelle est la complexité de votre fonction, toujours en fonction de la hauteur h de l'arbre ?

Question 4) On souhaite maintenant coder l'opération contraire, c'est-à-dire supprimer le dernier élément de l'ACG.

a) Décrivez le principe de cette suppression selon la signature de l'arbre.

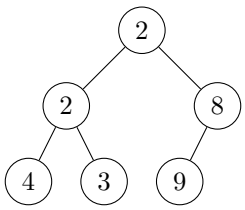
b) Écrivez une fonction `supprimer t` de type `arbre -> arbre` qui calcule à partir d'un ACG non vide l'ACG obtenu en supprimant la dernière feuille.

c) Quelle est sa complexité en fonction de h ?

Partie 3 - Tas-minimum

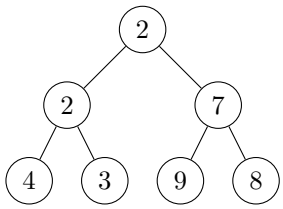
Dans cette partie, on ajoute une contrainte sur les ACG : on dit qu'un ACG est un tas quand toute valeur d'un sommet est inférieur ou égale à celles qui se trouvent dans son sous-arbre.

Un exemple de tas



Question 1) On modifie la façon d'ajouter un élément : on dira « insérer un élément dans un tas ». L'objectif est désormais double : il s'agit d'ajouter un élément dans l'ACG de façon à obtenir toujours un ACG comme précédemment, mais aussi à obtenir aussi un nouveau tas.

Dans le tas de l'exemple précédent, on insère le nombre 7 : on obtient alors le tas



Remarquez comment le 7 a forcé le déplacement du 8...

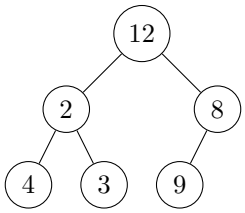
On procède de la façon suivante : soit S(x, b, fg, fd) le tas dans lequel on veut insérer l'entier y, alors
- si x > y, l'entier y doit être inséré à la racine de l'arbre, car il est plus petit que tous les éléments de l'arbre ; on remplace donc x par y et on réinsère x dans l'arbre ainsi calculé ;
- sinon l'entier y sera inséré en-dessous de x, ce qui est faisable par appel récursif, comme dans le cas de l'ajout traité précédemment.

a) Écrivez une fonction remplacer y t de type `int -> arbre -> arbre` qui calcule l'arbre obtenu à partir de t en remplaçant la valeur de la racine par y.

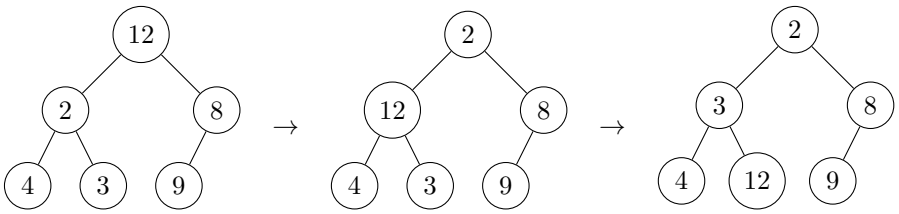
b) Écrivez une fonction inserer y t de type `int -> arbre -> arbre` qui insère l'entier y dans le tas t.

Question 2) Dans cette question, on suppose qu'on a un ACG qui est presque un tas : tous les éléments sont bien rangés, sauf la racine, qui est quelconque. Autrement dit, les deux fils de l'ACG sont deux tas, mais l'ACG n'est pas forcément un tas.

Un exemple de presque-tas



Pour rétablir la structure de tas, on fait progressivement descendre l'entier qui pose problème vers le fond en effectuant des échanges successifs.



Soit t = S(x, b, fg, fd) tel que les fils fg et fd soient des tas dont les racines ont pour valeurs xg et xd respectivement.

a) À quelle condition sur x, xg, xd peut-on affirmer que t est un tas ?

b) Dans le cas contraire, montrez que si on échange x avec l'un des deux xg ou xd, alors on peut récursivement construire un tas à partir de t.

c) Écrivez la fonction ordonner t de type `arbre -> arbre` qui réordonne un tas selon le processus précédent.

Question 3) La dernière opération importante est l'extraction du minimum d'un tas. En vous servant des fonctions dernier, supprimer, ordonner, justifiez qu'on peut calculer à partir d'un tas non vide un couple x,u où x est le minimum des éléments du tas et u un tas obtenu à partir de t en supprimant sa racine.

Question 4) Quelles sont les complexités de vos fonctions sur les tas ?

Problème 2 - Un mini-LaTeX

TeX est un système de composition de texte (ce devoir est écrit avec LaTeX, un dérivé de TeX) créé dans les années 70 par D. Knuth pour permettre d'écrire des articles, des thèses sur un simple ordinateur avec une qualité typographique proche de celle qu'on pouvait obtenir en confiant son texte à un imprimeur professionnel. Un des problèmes majeurs de la composition typographique est de savoir comment grouper les mots sur les lignes pour obtenir un résultat visuel le plus agréable.

Le principe est le suivant. On considère un texte constitué de n mots de longueur $\ell_0, \ldots, \ell_{n-1}$: un mot est une suite de symboles contigus sans espaces, ponctuation comprise, qu'on appelle les lettres. Par exemple, `bonjour!` est un mot.

Chaque lettre occupe la même largeur (ce qui n'est pas vrai en toute rigueur, mais nous faisons simple). Les mots sont séparés par des espaces uniques de même largeur que les lettres. Chaque ligne du paragraphe a une largeur fixe max : on peut donc considérer que les lignes sont des alignements de max cases dans lesquelles il va falloir placer les lettres et les espaces. On remplit les lignes par la gauche (pas d'espaces à gauche inutiles) et on suppose bien sûr que tous les mots sont de longueurs inférieures ou égales à max. On ne veut pas couper les mots, donc on finit les lignes par des espaces : le nombre d'espaces vides à la fin de chaque ligne est appelé médiocrité de la ligne. La médiocrité d'un paragraphe est la somme des carrés des médiocrités des lignes, sauf de la dernière, qui ne compte pas. L'objectif pour le système TEX est de savoir comment remplir les lignes pour minimiser la médiocrité du paragraphe.

Exemple.

Considérons le texte : `ce texte n'a aucun intérêt autre que celui d'être découpé en lignes pas trop médiocres.`

Si $max = 25$, on peut couper ce texte de plusieurs façons (les espaces de fin de lignes sont figurées [1] par des caractères de soulignements _).

```
ce texte n'a aucun_______
intérêt autre que________
celui d'être découpé en__
lignes pas trop médiocres.
```

Avec ce découpage, la médiocrité du paragraphe est de $7^2 + 8^2 + 2^2 = 117$.

```
ce texte n'a aucun_______
intérêt autre que celui__
d'être découpé en lignes_
pas trop médiocres.
```

Avec ce découpage, la médiocrité du paragraphe est de $7^2 + 2^2 + 1^2 = 54$.

On peut constater que le deuxième paragraphe est plus joli à regarder.

Pour $(i, j) \in [\![0, n-1]\!]^2$ tel que $i \leqslant j$, on note $s(i, j)$ la suite des mots numérotés de i à j séparés par une espace.

Question 1) Calculez le nombre de cases nécessaires pour contenir la suite de mots $s(i, j)$ (on appelle ça la longueur naturelle de la suite).

Question 2) Si le nombre précédent est plus petit que max, alors on peut calculer la médiocrité d'une ligne qui contient exactement la suite de mots $s(i, j)$ en fonction de max, i, j et les longueurs des mots. On la note $\text{med}(i, j)$. Donnez l'expression de $\text{med}(i, j)$.

Ce même nombre $\text{med}(i, j)$ a aussi un sens s'il est négatif : il signifie que la suite $s(i, j)$ est trop longue pour tenir sur une ligne de longueur max.

Question 3) Supposons que le système ait calculé le paragraphe optimal (celui de médiocrité minimale) sur m lignes ($m \geqslant 2$). Justifiez que les $m-1$ dernières lignes forment elles-mêmes un paragraphe optimal. Quel type de programmation peut-on mettre en œuvre ?

Question 4) On note $M(i)$ la médiocrité optimale du paragraphe composé avec la suite $s(i, n-1)$.

a) Que vaut $M(i)$ si $\text{med}(i, n-1) \geqslant 0$?

b) Si $\text{med}(i, n-1) < 0$, montrez que $M(i) = \min\limits_{\substack{i \leqslant k \leqslant n-1 \\ \text{med}(i,k) \geqslant 0}} \big(\text{med}(i, k)^2 + M(k+1)\big)$.

Question 5) Exemple : avec $max = 10$ et une suite de mots de longueurs $8, 1, 4, 2, 5$, calculez la médiocrité optimale du paragraphe et donnez un découpage associé.

Question 6) Donnez un algorithme qui calcule la médiocrité du paragraphe complet : elle prendra pour paramètre le tableau des longueurs des mots et le paramètre max.

Question 7) Donnez ensuite une version améliorée qui donne en plus le découpage du paragraphe en lignes : on calculera une liste (ou un tableau) de couples de la forme (i, j).

Remarque.

Dans la réalité, c'est plus compliqué. D'abord, les lettres n'ont pas toutes la même largeur, cela dépend de la police de caractère utilisée. Ensuite, les espaces sont compressibles ou étirables jusqu'à un certain point : le calcul de la médiocrité se fait en comparant la longueur naturelle à la longueur calculée (cela revient à répartir les espaces de fin de ligne de notre étude sur les espaces inter-mots : regardez attentivement les lignes de ce sujet, les espaces inter-mots n'ont pas toutes la même largeur). Enfin, un traitement supplémentaire est effectué : après avoir essayé de ne pas couper les mots, on a une solution, qui est analysée pour savoir si elle est belle selon d'autres critères (les espaces inter-mots entre deux lignes consécutives ne doivent pas être trop différents, par exemple). Si le paragraphe n'est pas assez beau, le système relance un calcul en autorisant les césures (les mots coupés en fin de ligne).

1. en typographie, le caractère blanc s'appelle une espace, c'est un mot féminin

Devoir surveillé 4 - Corrigé

Problème 1

Partie 1

Question 1)

```
let rec hauteur t =
  match t with
  | V -> -1
  | S(_,_,fg,fd) -> 1 + max (hauteur fg) (hauteur fd);;
```

Question 2) Soit $\mathscr{P}(h)$ la proposition « tout arbre binaire de hauteur h a au maximum 2^h feuilles ».

Si $h = -1$, l'arbre vide n'a pas de feuille et on a bien $0 \leqslant 2^{-1}$ donc $\mathscr{P}(-1)$ est vraie.

Si $h = 0$, alors un arbre de hauteur 0 n'a qu'un sommet qui est une feuille et on a bien $1 \leqslant 2^0$, donc $\mathscr{P}(0)$ est vraie.

Si $\mathscr{P}(-1), \ldots, \mathscr{P}(h-1)$ sont vraies, alors soit t un arbre de hauteur h : il a deux fils dont l'un est de hauteur $h-1$ et l'autre de hauteur au plus égale à $h-1$, donc par hypothèse de récurrence, ses deux arbres ont au plus 2^{h-1} feuilles. Comme la liste des feuilles de t est la concaténation des feuilles des fils gauche et droit, on en déduit que le nombre de feuilles de t est au plus égal à $2 \times 2^{h-1} = 2^h$, donc $\mathscr{P}(h)$ est vraie.

D'après le principe de récurrence forte, pour tout $h \geqslant -1$, $\mathscr{P}(h)$ est vraie.

Partie 2

Question 1) Si l'arbre est complet (cas D), alors ses fils le sont aussi donc l'arbre a la signature $(true, true, true)$.

Sinon le premier booléen est $false$. Quant aux deux autres, il y a trois cas :
- dans le cas A, on a la signature $(false, false, true)$;
- dans le cas B, on a la signature $(false, true, true)$;
- dans le cas C, on a la signature $(false, true, false)$;

On constate que dans chaque cas, on a une signature différente, donc la connaissance de la signature permet de connaître la catégorie.

Question 2)

a) Dans un parcours en largeur de l'arbre, le dernier élément du parcours est exactement le dernier élément de l'arbre.

b) Arbre A : 4 ; arbre B : 0 ; arbre C : 6 ; arbre D : 7.

c) On considère un arbre d'au moins deux feuilles (les arbres à une feuille sont les cas de base).
- Si l'arbre est de signature $(false, false, true)$, alors le dernier élément est dans le fils gauche, car celui-ci est strictement plus haut que le fils droit.
- Si l'arbre est de signature $(false, true, true)$, on a le même résultat.
- Si l'arbre est de signature $(false, true, false)$, alors le dernier élément est dans le fils droit, car il a la même hauteur que le fils gauche.
- Si l'arbre est complet, de signature $(true, true, true)$, alors le dernier élément est celui de son fils droit.

On constate que le dernier est à gauche si et seulement si le premier et troisième booléens sont différents.

d)
```
let rec dernier t =
  match t with
  | V -> failwith "arbre vide"
  | S(x, _, V, V) -> x
  | S(_, b, fg, fd) -> let bd = complet fd in
    if b = bd then dernier fd
    else dernier fg;;
```

e) La complexité de la fonction précédente vérifie la relation de récurrence $C(h) = C(h-1) + O(1)$ car il y au seul appel récursif sur un arbre de hauteur au plus $h-1$, donc la complexité est en $O(h)$.

Question 3)

a) On considère un arbre d'au moins une feuille (l'arbre vide est le cas de base).
- Si l'arbre est de signature $(false, false, true)$, alors le fils gauche n'est pas complet, l'élément à ajouter sera au bout du fils gauche.

— Si l'arbre est de signature $(false, true, true)$, alors le fils gauche est complet et le fils droit aussi, mais comme l'arbre n'est pas complet, le fils droit est de hauteur strictement inférieure à celle du fils gauche, donc on ajoute un dernier élément en ajoutant une nouvelle ligne au fils droit : on l'ajoute au fils droit.
— Si l'arbre est de signature $(false, true, false)$, alors le fils gauche est complet, mais le droit ne l'est pas, tout en ayant la même hauteur, donc on ajoute l'élément à droite.
— Si l'arbre est complet, de signature $(true, true, true)$, alors on doit ajouter une nouvelle ligne et on place l'élément à gauche.

On constate que l'ajout se fait à gauche si et seulement si les premier et deuxième booléens sont égaux.

b) Le seul cas où l'arbre peut devenir complet est le cas C, à la condition que le fils droit devienne lui-même complet : après ajout dans le fils droit, on regarde son booléen et on place le même dans la racine.

Dans les autres cas, le booléen de la racine est $false$.

c)
```
let rec ajouter x t =
  match t with
  | V -> S(x, true, V, V)
  | S(y, b, fg, fd) -> let bg = complet fg in
    if b = bg then
      S(y, false, ajouter x fg, fd)
    else
      let fd' = ajouter x fd in
      S(y, complet fd', fg, fd');;
```

d) On a le même type de relation de récurrence sur la complexité que pour la fonction précédente, encore une fois on obtient une complexité linéaire en la hauteur.

Question 4)

a) On retrouve les mêmes cas que dans le calcul du dernier élément, puisque c'est cet élément qui est supprimé. De plus, si on supprime un élément à droite, le booléen de la racine devient $false$; et dans le cas d'une suppression à gauche, si le nouvel arbre gauche est complet, alors le nouvel arbre l'est aussi et seulement dans ce cas, le booléen de la racine est donc égal à celui de son fils gauche.

b)
```
let rec supprimer t =
  match t with
  | V -> failwith "arbre vide"
  | S(_, _, V, V) -> V
  | S(y, b, fg, fd) -> let bd = complet fd in
    if b = bd then
      S(y, false, fg, supprimer fd)
    else
      let fg' = supprimer fg in
      S(y, complet fg', fg', fd);;
```

c) Toujours pareil !

Partie 3

Question 1)

a)
```
let remplacer x a =
  match a with
  | V -> failwith "arbre vide"
  | S(_, b, fg, fd) -> S(x, b, fg, fd);;
```

```
let rec inserer x t =
  match t with
  | V -> S(x, true, V, V)
  | S(y, b, fg, fd) ->
     if x < y then inserer y (remplacer x t)
     else let bg = complet fg in
      if b = bg then
        S(y, false, inserer x fg, fd)
      else let fd' = inserer x fd in
        S(y, complet fd', fg, fd');;
```

Question 2)

a) t est un tas si et seulement si on a x < xg et x < xd.

b) Si on n'est pas dans le cas précédent, alors on échange x avec le minimum de xg et xd : disons par exemple que c'est xg. On place alors xg à la racine : c'est le plus petit élément du tas, on remplace xg dans le fils gauche par x et on réordonne cet arbre, le fils droit n'ayant pas changé.

Bien sûr, si l'un des deux fils est vide (ce ne peut être que le fils droit), alors on ne fait que comparer x avec xg.

c)

```
let rec ordonner t =
  match t with
  | V | S(_,_,_,V,V) -> t
  | S(x, b, fg, V) -> let xg = valeur fg in
     if xg >= x then t else S(xg, b, S(x, true, V, V), V)
  | S(x, b, fg, fd) -> let xg = valeur fg and xd = valeur fd in
     if x > max xg xd then
       if xg < xd then S(xg, b, ordonner (remplacer x fg), fd)
       else S(xd, b, fg, ordonner (remplacer x fd))
     else if x > xg then S(xg, b, ordonner (remplacer x fg), fd)
     else if x > xd then S(xd, b, fg, ordonner (remplacer x fd))
     else t;;
```

Question 3)

```
let extraire t =
  let x = valeur t in
  let d = dernier t in
  let t' = supprimer t in
  if t' <> V then
    let t'' = remplacer d t' in
    x, ordonner t'
  else x,V;;
```

Question 4) La fonction remplacer est de complexité constante, les complexités de inserer et ordonner suivent la même relation de récurrence que précédemment, donc elles sont de complexité linéaire. Enfin, extraire utilise successivement ces fonctions, donc elle est elle-même de complexité linéaire en h.

Problème 2

Question 1) La longueur naturelle est la longueur des mots, plus celle des espaces intermédiaires : $\sum_{k=i}^{j} \ell_k + (j-i)$.

Question 2) Donc $\text{med}(i,j) = max - \sum_{k=i}^{j} \ell_k - (j-i)$.

Question 3) Par l'absurde : si les $m-1$ dernières lignes ne sont pas coupées optimalement, alors on choisit un découpage optimale, on lui ajoute la première ligne et on obtient un découpage total de médiocrité plus petite : contradiction.

Ceci est l'expression du principe de sous-optimalité de Bellmann. On va donc utiliser les techniques de programmation dynamique.

Question 4)

a) Si $\text{med}(i, n-1) \geqslant 0$, alors les mots de i à $n-1$ tiennent sur une ligne, la dernière, donc $M(i) = 0$.

b) Sinon, on cherche le mot qui suit le mot i en lequel on va couper la ligne (médiocrité de la ligne positive) et qui donne une médiocrité totale minimale : application du principe de sous-optimalité de Bellmann.

Question 5) On veut calculer $M(0)$. On doit donc calculer $M(0) = \min_{\substack{0 \leqslant k \leqslant 3 \\ \text{med}(0,k) \geqslant 0}} \big(\text{med}(0,k)^2 + M(k+1)\big)$.

On constate aisément que dès qu'on prend plus de trois mots, on obtient une première ligne trop longue, donc $M(0) = \min_{0 \leqslant k \leqslant 1} \big(\text{med}(0,k)^2 + M(k+1)\big)$.

Par appel récursif, on doit maintenant calculer $M(1)$ et $M(2)$.

De même, pour calculer $M(1)$, on va tenter de faire une ligne avec les mots numérotés 1, 2 et 3 au maximum (avec les mots suivants c'est trop longs), donc $M(1) = \min_{1 \leqslant k \leqslant 3} \big(\text{med}(1,k)^2 + M(k+1)\big)$, donc par appel récursif, on calcule $M(2)$, $M(3)$, $M(4)$.

Etc.

Pour calculer concrètement, mettons-nous à la place de l'ordinateur, qui une fois arrivé sur les cas de base, remonte les calculs.

On va donc commencer par calculer $M(4)$, puis $M(3)$, $M(2)$, $M(1)$ et $M(0)$.

$M(4) = 0$ car le mot numéro 4, de longueur 5, tient sur une ligne.

$M(3) = 0$ car les deux mots numéros 3 et 4 tiennent sur une ligne.

En revanche, $M(2)$ n'est pas nul, car les trois mots numéros $2, 3, 4$ sont trop longs pour tenir sur une ligne : on peut couper au mot 2 ou 3, on calcule alors $\text{med}(2,2) = 10-4 = 6$ et $\text{med}(2,3) = 10-(4+2)-1 = 3$, donc $M(2) = \min(6^2+0, 3^2+0) = 9$: on coupe au mot 3 dans ce cas.

On fait de même pour $M(1)$: on peut couper au mot 1 ou 2 ou 3, puis on calcule $\text{med}(1,1) = 10-1 = 9$, $\text{med}(1,2) = 10-(1+4)-1 = 4$, $\text{med}(1,3) = 10-(1+4+2)-2 = 1$, donc $M(1) = \min(9^2+9, 4^2+0, 1^2+0) = 1$

Enfin, on calcule $\text{med}(0,0) = 10-8 = 2$ et $\text{med}(0,1) = 10-(8+1)-1 = 0$, donc $M(0) = \min(2^2+1, 0^2+9) = 5$.

Si on suit les calculs, on retrouve les minimums et on sait où on coupe : la première ligne n'aura que le mot 0, la seconde aura les mots $1, 2, 3$ et la dernière aura le mot 4.

Question 6)

```
(* calcul de mediocrite d'une ligne entre les mots i et j *)
let med i j l max =
   let s = ref 0 in
   for k = i to j do
      s := !s + l.(k)
   done;
   max - !s - j + i;;

let tex max l =
 let n = vect_length l in
 let m = make_vect (n) (-1) in
 let rec taux i =
   if m.(i) <> -1 then m.(i)    (* 1 *)
   else let r =
     let mediocrite = med i (n-1) l max in
     if mediocrite >= 0 then (0)     (* 2 *)
     else begin      (* 3 *)
       let rec mini k m =
         if k = n then m
         else
           let medio = med i k l max in
           let m' = if medio >= 0 then
             min (medio * medio + taux (k+1)) m
           else m
         in mini (k+1) m'
       in
       let mi = med i i l max in
       mini (i+1) (mi * mi + taux (i+1))
     end
   in
   m.(i) <- r;    (* 4 *)
   r
 in
 taux 0;;
```

Question 7)

```
let tex max l =
 let n = vect_length l in
 let m = make_vect (n) (-1,0) in
 let med i j =
   let s = ref 0 in
   for k = i to j do
      s := !s + l.(k)
```

```
    done;
    max - !s - j + i
  in
  let rec taux i =
    if fst m.(i) <> -1 then m.(i)
    else let r =
      let mediocrite = med i (n-1) in
      if mediocrite >=0 then (0,n-1)
      else begin
        let mi = med i i and ti = taux (i+1) in
        let mini = ref (mi * mi + fst ti, 0) in
        for k = (i + 1) to (n-1) do
          let mk = med i k in
          if mk >=0 then
            let tk = taux (k+1) in
            let tmp = mk * mk + fst tk in
            if fst !mini > tmp then mini := (tmp, k)
        done;
        !mini
      end
    in
    m.(i) <- r;
    r
  in
  taux 0, m;;

let decoupe m =
  let n = vect_length m in
  let l = ref [] in
  let li = ref 0 in
  while !li <= (n-1) do
    let medio, suiv = m.(!li) in
    l := (!li,suiv) :: !l;
    li := suiv + 1
  done;
  rev(!l);;
```

La première fonction été modifiée pour donner en plus le tableau-mémoire, qui contient les médiocrités calculées mais aussi l'endroit où on coupe chaque ligne.

La seconde fonction reprend ce tableau-mémoire et récupère les endroits de coupure pour les assembler en couples.