Corrigé DM Option Info N°2

1) Problème 1

```
let rec longueur = function
  [] -> 0
  | a::r -> 1 + longueur r
```

```
let rec insertion l e = match l with
  | [] -> [e]
  | a::r when a < e -> a::(insertion r e)
  | _ -> e::l
```

```
let rec tri_insertion = function
  | [] -> []
  | a::r -> insertion (tri_insertion r) a
```

```
let selection_n l n =
  let lt = tri_insertion l in
  let rec aux l = function (* k-ième élément d'une liste *)
    | 0 -> List.hd l
    | k -> aux (List.tl l) (k-1) in
  aux lt (n-1)
```

```
let rec paquets_de_cinq l =
  let rec aux k = function (* Extrait k éléments de la liste, si possible *)
    | [] -> ([], [])
    | a::r -> let (u, v) = aux (k-1) r in (a::u, v) in
  let (p, r) = aux 5 l in
  p::(paquets_de_cinq r)
```

```
(* Ou *)
let rec paquets_de_cinq = function
  | a::b::c::d::e::r -> [a;b;c;d;e]::(paquets_de_cinq r)
  | l -> [l]
```

```
let rec medians = function
  | [] -> []
  | p::r -> let n = List.length p in
    (* Selection prend un indice ≥ 1 *)
    (selection_n p (n/2 + 1))::(medians r)
```

```
let rec partage p = function
  | [] -> ([], [], 0, 0)
  | a::r -> let (l1, l2, n1, n2) = partage p r in
    if a <= p then (a::l1, l2, n1+1, n2)
    else (l1, a::l2, n1, n2+1)
```

```
let rec selection l k =
  let n = List.length l in
  (* L'énoncé n'est pas cohérent entre selection_n et selection *)
  if n <= 5 then selection_n l (k+1) else begin
    let l_cinq = paquets_de_cinq l in
    let m = medians l_cinq in
    let pivot = selection m (((n+4)/5)/2) in
    let (l1, l2, n1, n2) = partage pivot l in
    if k < n1 then selection l k else selection l2 (k-n1)
  end
```

Correction On admet la relation de récurrence $\forall n \geq 55, T(n) \leq T\left(\left\lfloor \frac{n+4}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{8n}{11} \right\rfloor\right) + 4n$.

On montre que pour tout entier $n \geq 1$, on a $T(n) \leq (200 + T(55))n$, par récurrence forte.

Notons que la fonction T est croissante, la relation est bien vérifiée pour tout $n \leq 55$.

Soit $n \geq 55$, on suppose que la relation est vérifiée pour tout $k \in \llbracket 1, n \rrbracket$. D'après la relation de récurrence, on a

$$T(n) \leq T\left(\left\lfloor \frac{n+4}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{8n}{11} \right\rfloor\right) + 4n \leq (200 + T(55))\left(\left\lfloor \frac{n+4}{5} \right\rfloor + \left\lfloor \frac{8n}{11} \right\rfloor\right) + 4n.$$

On a $\lfloor \frac{n+4}{5} \rfloor + \lfloor \frac{8n}{11} \rfloor \leq \frac{n+4}{5} + \frac{8n}{11} = \frac{4}{5} + \frac{51}{55}n$. On en déduit que

$$T(n) \leq (200 + T(55))n - \frac{4}{55}n(200 + T(55)) + 4n + \frac{4}{5} \leq (200 + T(55))n - \frac{800}{55}n + 4n + 1 \leq (200 + T(55))n.$$

2) Problème 2

Correction Dans le cas d'une liste vide, fonction1 renvoie un tableau d'entier, de type `int array`. Dans le cas général, l'appel récursif `let theta = ...` renvoie donc un tableau d'entiers, et c'est cette quantité qui est finalement renvoyée.

Elle renvoie un tableau `t` de taille λ tel que, pour tout $i \in \llbracket 0, \lambda - 1 \rrbracket$, `t[i]` soit le nombre d'occurrences de la lettre σ_i dans le texte.

Démonstration : Par récurrence sur la taille $|t|$ du texte.

- Pour un texte vide, le tableau renvoyé ne contient que des 0.
- Si on suppose le résultat vrai pour tout texte de taille $n \in \mathbb{N}$ fixée, et qu'on considère un texte `t`, avec $|t| = n + 1$, alors le texte `t` peut s'écrire `t = u : s`, où `s` est un texte de taille n , qui relève de l'hypothèse de récurrence. Dans ce cas, fonction1 calcule le tableau des occurrences de `s` (correctement, par hypothèse de récurrence), et incrémente la case correspondant à la lettre `u`. Elle renvoie donc bien le tableau des occurrences de `t`.

```
let cree_repartition t =
  let occs = fonction1 t in
  let rec aux i = match i with
    | i when i = lambda -> []
    | i -> let r = aux (i+1) in if occs.(i) = 0 then r else ((i, occs.(i)))::r in
  aux 0
```

Correction La fonction1 a une complexité temporelle linéaire en la longueur du texte (+ la création d'un tableau de taille λ), et en plus de son appel à fonction1, cree_repartition parcourt le tableau occs de taille λ . Donc cree_repartition a une complexité en $O(\lambda + |t|)$.

La complexité en espace de cree_repartition est en $O(|\lambda|)$, pour l'allocation du tableau d'occurrences, et l'allocation d'une nouvelle liste, de longueur au plus $|\lambda|$.

La liste renvoyée par cree_repartition a une taille qui est d'une part inférieure à $|\lambda|$, et d'autre part inférieure à $|t|$, donc en $O(\min(|\lambda|, |t|))$.

Pour un courriel de la vie courante encodé en unicode, l'énoncé donne $|\lambda| = 1114111$, et la taille du texte est de l'ordre de quelques centaines. Mieux que ça, le nombre de caractères distincts utilisés est de l'ordre d'au plus 50. La représentation de l'ensemble des caractères du texte par sa répartition plutôt que par le tableau des occurrences est donc intéressante.

```
(* r est une list de couples, u est un entier *)
let rec incremente_repartition l u = match l with
| [] -> [(u, 1)]
| (v, c)::r when v = u -> (v, c+1)::r
| a::r -> a::(incremente_repartition r u)
```

```
let cree_modulaire t m =
  let table = Array.make m [] in
  let rec aux = function (* Prend en argument le reste du texte *)
    | [] -> ()
    | a::r -> let j = a mod m in
              table.(j) <- incremente_repartition table.(j) a in
  aux t;
  table
```

```
let valence table =
  let v = ref 0 in
  for i=0 to Array.length table do
    v := !v + List.length table.(i)
  done;
  !v
```

Correction QUESTION 10 La notion d'espérance conditionnelle n'est pas au programme. On se permet certaines manipulations naturelles.

Si $\ell = \ell_0$, on a

$$\begin{aligned} \mathbf{E}[Z_\ell \mid X_{i_0} = \ell_0] &= \mathbf{E}[|\{i \in \llbracket 1, v \rrbracket; X_i = \ell\}| \mid X_{i_0} = \ell] = \mathbf{E}[1 + |\{i \in \llbracket 1, v \rrbracket \setminus \{i_0\}; X_i = \ell\}| \mid X_{i_0} = \ell] \\ &= \mathbf{E}[1 + |\{i \in \llbracket 1, v \rrbracket \setminus \{i_0\}; X_i = \ell\}|] = 1 + \mathbf{E}[|\{i \in \llbracket 1, v \rrbracket \setminus \{i_0\}; X_i = \ell\}|], \end{aligned}$$

où l'avant dernière étape est justifiée par l'indépendance de la variable dont on prend l'espérance et de X_{i_0} .

La variable $|\{i \in \llbracket 1, v \rrbracket \setminus \{i_0\}; X_i = \ell\}|$ compte un nombre de succès, lors de $v - 1$ épreuves de Bernoulli indépendantes, de paramètre $\frac{1}{m}$. Elle suit donc une loi binomiale $\mathcal{B}(v - 1, \frac{1}{m})$, qui a pour espérance $\frac{v-1}{m}$.

On en déduit que $\mathbf{E}[Z_\ell \mid X_{i_0} = \ell_0] = 1 + \frac{v-1}{m}$.

Quand $\ell \neq \ell_0$, le même raisonnement donne $\mathbf{E}[Z_\ell \mid X_{i_0} = \ell_0] = \frac{v-1}{m}$.

Correction QUESTION 11 La notion de complexité moyenne en temps n'est pas au programme. Elle est à comprendre ici comme l'espérance du nombre d'opérations, si les données d'entrées sont des variables aléatoires suivant les lois spécifiées (implicitement...) par l'énoncé.

La complexité de `incremente_repartition` est (au pire des cas) linéaire en la longueur de la répartition.

Dans le contexte de `creer_modulaire`, on fait $|t|$ appels à cette fonction `incremente_repartition`, sur une des m répartitions.

L'énoncé impose que le texte admette exactement v lettres distinctes.

On modélise cela en imaginant que les caractères du textes sont des entiers de $\llbracket 0, v-1 \rrbracket$. Pour $i \in \llbracket 1, v \rrbracket$, on convient que le reste modulo m du vrai numéro de la lettre i est noté X_i , qui suit une loi uniforme sur $\llbracket 0, m-1 \rrbracket$ (c'est douteux, cela suppose au moins que m divise le nombre total de lettres). On convient également que les X_i sont indépendantes.

Alors, avec les notations de la question précédente, pour $\ell \in \llbracket 0, m-1 \rrbracket$, Z_ℓ représente la longueur finale de la répartition d'indice ℓ . Chaque caractère c du texte va contribuer à la complexité la valeur Z_{X_c} . On peut donc écrire que la complexité totale est majorée par

$$O\left(|m| + |t| + \sum_{c \in t} Z_{X_c}\right).$$

D'après la question précédente, on a

$$\begin{aligned} \mathbf{E}(Z_{X_c}) &= \sum_{\ell=0}^{m-1} \mathbf{E}(Z_{X_c} \mid X_c = \ell) P(X_c = \ell) = \sum_{\ell=0}^{m-1} \mathbf{E}(Z_{X_c} \mid X_c = \ell) P(X_c = \ell) \\ &= \sum_{\ell=0}^{m-1} \mathbf{E}(Z_\ell \mid X_c = \ell) P(X_c = \ell) = \sum_{\ell=0}^{m-1} \left(1 + \frac{(v-1)}{m}\right) \frac{1}{m} \\ &= 1 + \frac{(v-1)}{m} \end{aligned}$$

On en déduit que la complexité moyenne est de $O\left(|m| + |t| + |t| + \frac{(v-1)|t|}{m}\right)$.

Correction QUESTIONS 12, 13 L'appel `creer_modulaire t m` a une complexité en espace égale à la somme des tailles des listes dans le tableau `theta`, cette somme vaut exactement la valence du texte, donc la complexité est en $O(m+v)$ (m correspondant à l'allocation du tableau lui-même).

Si l'on prend m de l'ordre de la valence du texte, c'est-à-dire de l'ordre de 26, la complexité spatiale précédente devient $O(v)$, et la complexité temporelle devient $O(|t|)$, ces expressions étant optimales.

a) Encodeur optimal

```
let rec codage_uni (u : unicode) (code : code) = (* renvoie l'encodage du caractère *)
  match code with
  | Nil -> failwith "Code non présent dans l'arbre"
  | Noeud (uu, l, c1, c2) when u = uu -> l
  | Noeud (uu, l, c1, c2) when u > uu -> codage_uni u c2
  | Noeud (uu, l, c1, c2) -> codage_uni u c1
let rec encodeur t code = match t with
  | [] -> []
  | u::r -> (codage_uni u)@(encodeur r)
```

Correction Pour encoder un caractère u , la complexité de `codage_uni` est en $\text{prof}_{\mathcal{A}}(u)$.

Dans `encodeur`, l'opération de concaténation `@` a une complexité linéaire en la longueur de l'encodage de u , donc majorée par $L = \max_{\sigma \in \Sigma} |c(\sigma)|$.

Chaque caractère du texte contribue donc $\text{prof}_{\mathcal{A}}(u) + L$ à la complexité totale. La complexité totale est $|t| \cdot L + \sum_{\sigma \in \Sigma} |t|_{\sigma} \text{prof}_{\mathcal{A}}(\sigma)$.

Correction On a, en notant ρ la racine de $\mathcal{A}$

$$\begin{aligned} \text{Prof}(\mathcal{A}) &= \sum_{\sigma \in \Sigma} f(\sigma) \text{prof}_{\mathcal{A}}(\sigma) = f(\rho) + \sum_{\sigma \in \Sigma \mid \sigma \in \mathcal{A}_g} f(\sigma) \text{prof}_{\mathcal{A}}(\sigma) + \sum_{\sigma \in \Sigma \mid \sigma \in \mathcal{A}_d} f(\sigma) \text{prof}_{\mathcal{A}}(\sigma) \\ &= f(\rho) + \sum_{\sigma \in \Sigma \mid \sigma \in \mathcal{A}_g} f(\sigma)(1 + \text{prof}_{\mathcal{A}_g}(\sigma)) + \sum_{\sigma \in \Sigma \mid \sigma \in \mathcal{A}_d} f(\sigma)(1 + \text{prof}_{\mathcal{A}_d}(\sigma)) \\ &= \sum_{\sigma \in \Sigma} f(\sigma) + \sum_{\sigma \in \Sigma \mid \sigma \in \mathcal{A}_g} f(\sigma) \text{prof}_{\mathcal{A}_g}(\sigma) + \sum_{\sigma \in \Sigma \mid \sigma \in \mathcal{A}_d} f(\sigma) \text{prof}_{\mathcal{A}_d}(\sigma) \\ &= \sum_{\sigma \in \Sigma} f(\sigma) + \text{Prof}(\mathcal{A}_g) + \text{Prof}(\mathcal{A}_d) \end{aligned}$$

Correction QUESTION 17 On a $\Pi_{u,u} = f(u)$, réalisé pour un arbre avec un unique nœud.

Soient $u < v$. D'après la définition d'un arbre de code, si un arbre $\mathcal{A}$ représente $\Sigma_{u,v}$, alors il existe $r \in \llbracket u, v \rrbracket$ tel que σ_r soit la racine de $\mathcal{A}$, $\Sigma_{u,r-1}$ soit représenté par $\mathcal{A}_g$ et $\Sigma_{r+1,v}$ soit représenté par $\mathcal{A}_d$. Si $\mathcal{A}$ est optimal, nécessairement, $\text{Prof}(\mathcal{A}_g) = \Pi_{u,r-1}$ et $\text{Prof}(\mathcal{A}_d) = \Pi_{r+1,v}$, donc la profondeur de $\mathcal{A}$ vaut $\sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \Pi_{r+1,v} + \Pi_{u,r-1}$.

La profondeur optimale pour $\Sigma_{u,v}$ est obtenue en prenant le minimum, sur r , de ces quantités. Elle vaut donc

$$\Pi_{u,v} = \sum_{\sigma \in \Sigma_{u,v}} f(\sigma) + \min_{r \in \llbracket u,v \rrbracket} (\Pi_{r+1,v} + \Pi_{u,r-1}),$$

avec la convention $\Pi_{v+1,v} = \Pi_{u,u-1} = 0$.

Correction QUESTION 18 Il s'agit de calculer, par un paradigme de programmation dynamique, trois quantités qui dépendent de $u \leq v \in \llbracket 0, n \rrbracket$: les $\Pi_{u,v}$, les arbres optimaux correspondant $\mathcal{A}_{u,v}$ et les $S_{u,v} = \sum_{\sigma \in \Sigma_{u,v}} f(\sigma)$.

On stocke ces trois données dans trois tableaux à deux dimensions, de tailles $\lambda \times \lambda$, initialisés à des valeurs nulles pour Π , des arbres Nil pour $\mathcal{A}$ et des sommes nulles pour S .

On commence par initialiser les trois diagonales de ces tableaux, avec $\Pi_{u,u} = f(u)$, $S_{u,u} = f(u)$, et $\mathcal{A}_{u,u}$ un arbre Noeud(u , $c(u)$, Nil, Nil).

Puis, pour r variant de 1 à λ , et pour u variant de u à $\lambda - r$, on effectue le calcul des valeurs d'indices $(u, u+r)$ des trois tableaux.

On commence par calculer $S_{u,u+r}$ en $O(1)$. Puis, par une recherche de minimum, en utilisant les valeurs déjà calculées, on trouve $\Pi_{u,u+r}$, et l'arbre $\mathcal{A}_{u,u+r}$. Cette recherche de minimum a une complexité en $O(\lambda)$.

Finalement, la complexité est en $O(\lambda \times \lambda \times \lambda)$.

3) Problème 3

a) Flots

```
let rec compte_feuilles = fonction
  | Feuille -> 1
  | Interne (g, d) -> compte_feuilles g + compte_feuilles d
```

```
let compatible arb flot =
  (* La fonction aux prend en argument un arbre, le numéro de la
  * prochaine feuille, et renvoie un triplet (b, s, t), où b est un
  * booléen qui dit si le flot est compatible et s est le numéro
  * associé à la racine de l'arbre et t est le nb de feuilles de l'arbre *)
  let rec aux arb i = match arb with
    | Feuille -> (true, flot.(i), 1)
    | Interne (g, d) ->
      let (b1, s1, n1) = aux g i in
      if not b1 then (false, 0, 0) else
        let (b2, s2, n2) = aux d (i + n1) in
        if b1 && b2 && s1 + s2 mod 4 <> 0 then
          (true, s1 + s2 mod 4, n1 + n2)
        else
          (false, 0, 0) in
  let (b, _, _) = aux arb 0 in
  b
```

Correction QUESTION 3 1. Il s'agit de montrer qu'un arbre à n feuilles admet $n - 1$ nœuds internes.

Par double comptage : le nombre d'arêtes de l'arbre est d'une part égale à deux fois le nombre de nœuds internes, car chaque nœud interne a deux fils (l'arbre est strict), et d'autre part égale au nombre de nœuds différents de la racine, puisque chaque tel nœud a un unique parent.

On en déduit que $2n_i = n_i + n_f - 1$, donc que $n_i = n_f - 1$.

2. On sait que $f(a) \equiv f(g) + f(d)[4]$, et qu'aucun de ces trois nombres n'est nul.

Pour $a = 1$, les couples sont (2, 3) et (3, 2).

Pour $a = 2$, les couples sont (1, 1) et (3, 3).

Pour $a = 3$, les couples sont (1, 2) et (2, 1).

Il y a donc deux couples, pour chaque a .

3. Si a est une feuille, on a $F(a, v) = 1$. Si $a = (g, d)$ et $(v_1, v_2), (v'_1, v'_2)$ sont les deux couples possibles pour les flots de g et d , on a $F(a, v) = F(g, v_1)F(d, v_2) + F(g, v'_1)F(d, v'_2)$.

À partir de cette relation de récurrence, on montre par induction structurelle que $F(a, v)$ ne dépend pas de $v \in \{1, 2, 3\}$: c'est bien le cas pour une feuille, et si c'est vrai pour deux arbres g, d , c'est vrai pour l'arbre (g, d) .

On peut donc réécrire la relation de récurrence comme $F(a, v) = 2F(g, v)F(d, v)$.

Remarquons que le nombre de nœuds internes de l'arbre a , noté $n_i(a)$ vérifie $n_i(a) = 1 + n_i(d) + n_i(g)$, et que pour une feuille f , on a $F(f, v) = 1 = 2^{n_i(f)}$. On en déduit par une induction structurelle immédiate que $F(a, v) = 2^{n_i(a)}$, puis finalement que le nombre de flots compatibles avec a vaut $3 \cdot 2^{n_i(a)} = 3 \cdot 2^{n-1}$.

Correction QUESTION 4 1. Les flots pour lesquels $N_+(a, f) = 1$ sont les flots interrompus après un seul calcul de somme.

Le premier calcul de somme a lieu quand on somme les flots de deux feuilles d'un même parent. Ces deux feuilles f_1, f_2 sont définies comme les deux fils du premier nœud interne traité dans un parcours postfixe.

Le flot est incompatible si la somme donne 0, Autrement dit, si $f(f_1) \equiv -f(f_2)[4]$.

Le nombre de tels flots est de $3 \cdot 3^{n-2} = 3^{n-1}$, indépendamment de la forme de l'arbre a .

2. On considère le premier nœud interne traité dans l'ordre postfixe, noté n_i , dont les deux fils sont des feuilles f_1, f_2 , et l'arbre a' obtenu en remplaçant dans a ce nœud par une feuille f . L'arbre a' a donc $n - 1$ feuilles.

À chaque flot de a' , on peut associer exactement deux flots de a qui auront la même valeurs sur tous les nœuds de a , sauf les deux feuilles retirées. Ces deux flots sont ceux obtenus en gardant les mêmes valeurs aux feuilles différentes de f , et en choisissant les deux possibilités pour le couple $(f(f_1), f(f_2))$ qui vérifient $f(f_1) + f(f_2) \equiv f(f)$.

Réciproquement, tout flot de a qui n'est pas incompatible dès la première étape attribue une valeur non nulle à n_i , donc donne un flot de a' , qui a exactement deux tels antécédents par cette construction.

On obtient clairement, pour tout $k \geq 2$, $\nu_k(a) = 2\nu_{k-1}(a')$.

On montre par récurrence sur $n \geq 2$ que pour $a \in A_n$, et $k \in \llbracket 1, n - 1 \rrbracket$ on a $\nu_k(a) = 2^{k-1}3^{n-k}$. Pour $k \geq n$, $\nu_k(a) = 0$, puisque d'après une question précédente, l'arbre n'a que $n - 1$ nœuds internes, donc l'algorithme ne fait que $n - 1$ additions, au plus.

La récurrence ne pose pas de difficulté.

3. On a

$$\begin{aligned} \frac{1}{3^n} \sum_f N_+(a, f) &= \frac{1}{3^n} \left(\sum_{k=1}^{n-1} k\nu_k(a) + (n-1) \cdot 3 \cdot 2^{n-1} \right) \\ &= (n-1) \left(\frac{2}{3} \right)^{n-1} + \frac{1}{3^n} \sum_{k=1}^{n-1} k 2^{k-1} 3^{n-k} \\ &= (n-1) \left(\frac{2}{3} \right)^{n-1} + \frac{1}{3} \sum_{k=1}^{n-1} k \left(\frac{2}{3} \right)^{k-1}. \end{aligned}$$

D'après la formule de l'énoncé, cette dernière somme vaut $\frac{1 - \left(\frac{2}{3}\right)^{n-1} (n - (n-1)2/3)}{(1/3)^2}$.

La limite, quand $n \rightarrow +\infty$, de cette quantité est $\frac{1}{3} \frac{1}{(1/3)^2} = 3$.

Rmq : Le résultat est cohérent : en moyenne, on détecte rapidement que le flot est incompatible. L'hypothèse de l'énoncé que $N_+(a, f)$ soit une bonne mesure de la complexité de compatible n'est pas raisonnable, la complexité est finalement plutôt dominée par le temps du parcours pour trouver les nœuds correspondant, disons, aux quelques premières additions. Dans le meilleur des cas, pour un arbre équilibré, elle serait en $O(\ln n)$.

b) Triangulation

```
(* Est-ce que les segments s'intersectent ? Ils sont supposés distincts *)
let intersecte (a, b) (c, d) =
  if a <= c && d <= b then false
  else if (c > a && c < b) || (d > a && d < b) then true
  else false

let triangulation n segs =
  let rec intersecte_aucun a = function
    | [] -> true
    | b::r -> (not (intersecte_aucun b r)) && intersecte_aucun a r in

  let rec aux = function
    | [] -> true
    | u::r -> (intersecte_aucun u r) && aux r in

  List.length segs = n && aux segs
```

Correction QUESTION 7 On prend un arbre ayant pour racine le segment $(1, 8)$, avec deux fils : $(1, 6)$ et $(6, 8)$, le segment $(6, 8)$ ayant comme feuilles $(6, 7)$ et $(7, 8)$.

Le segment $(1, 6)$ a $(1, 3)$ et $(3, 6)$ comme fils, $(1, 3)$ a pour fils $(1, 2)$ et $(2, 3)$, etc.

En général, on prend pour racine de l'arbre le segment $(1, 8)$, qui appartient à une face triangulaire, avec comme deux autres côtés des segments $(1, a)$ et $(a, 8)$. Les segments de la triangulation peuvent alors être partitionnés en deux parties, suivant si leurs extrémités sont dans $\llbracket 1, a \rrbracket$, ou dans $\llbracket a, 8 \rrbracket$. On peut alors, par des appels récursifs, construire un arbre à partir de chacune des deux parties, qui seront les deux fils de la racine.

```

let rec partitionne (a, b) = function
| [] -> ([], [])
| (c, d)::r -> let (l1, l2) = partitionne (a, b) r in
                if (a < c && c < b) || (a < d && d < b) then
                    ((c,d)::l1, l2)
                else if (b < c || d < a) then (l1, (c,d)::l2)
                else (l1, l2)

(* Renvoie le maximum des b tels que (a, b) est dans l *)
let rec max_snd a = function
[] -> min_int
| (u, b)::r when u = a -> max b (max_snd a r)
| _::r -> max_snd a r

let rec retire a = function
| [] -> []
| b::r when b=a -> r
| b::r -> b::(retire a r)

let triangle_arbre n t =
let rec aux a b t =
    if b = a+1 then Feuille else
        let c = max_snd a t in
        let (l1, l2) = partitionne (a, c) t in
        let (a1, a2) = (aux a c l1, aux c b l2) in
        Interne (a1, a2) in
aux 1 n (retire (1,n) t)

```

```

let rec arbre_triangle arbre =
(* renvoie la liste des segments associés à l'arbre, sachant que la
première feuille correspond à un segment (a, a+1). Renvoie aussi le
nb de feuilles de l'arbre *)
let rec aux a arbre = match arbre with
| Feuille -> [(a, a+1)], 1
| Interne(g, d) -> let (l, n) = aux a g in
                    let (l1, m) = aux (a+n) d in
                    ((a, a+n)::(a+n, a+n+m)::l@l1, n + m) in
aux 1 arbre

```