DM 1 : corrigé
Option informatique

I Arbretas (en anglais : Treap = Tree + Heap)

On peut montrer qu'un arbre binaire de recherche (ABR) construit en ajoutant un à un n entiers choisis « uniformément au hasard » a une hauteur moyenne $O(\log(n))$.
Si les éléments à rajouter ne sont pas générés aléatoirement, mais sont tous connus à l'avance, on peut commencer par les mélanger aléatoirement puis les rajouter dans cet ordre aléatoire pour obtenir à nouveau une hauteur moyenne $O(\log(n))$. Pour cela, on considère l'algorithme suivant (mélange de Knuth), où `Random.int (i+1)` renvoie un entier uniformément au hasard entre 0 et `i` :

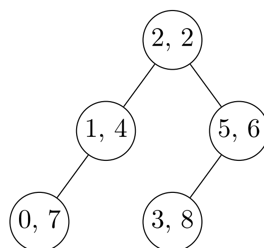
```
let shuffle t =
  for i = 0 to Array.length t - 1 do
    swap t i (Random.int (i+1))
  done;;
```

1. Écrire la fonction `swap : 'a array -> int -> int` utilisée par `shuffle`, telle que `swap t i j` échange `t.(i)` et `t.(j)`.
   ▶ Déjà faite en cours.
2. (Facultatif) Montrer que `shuffle t` applique une permutation choisie uniformément au hasard sur le tableau `t`.
   ▶ `shuffle t` applique un certain nombre de transpositions sur `t`, donc il effectue bien une permutation sur `t`.
   Soit σ une permutation. On sait que σ est produit de transpositions donc on peut l'écrire $\sigma = (a_1 b_1)(a_2 b_2)...(a_p b_p)$ avec $a_k < b_k$ pour tout k et $b_1 < b_2 < ... < b_p$. Alors il existe exactement une possibilité pour que `shuffle t` applique σ sur `t` : que `Random.int (i+1)` renvoie a_1 pour $\texttt{i} = b_1$ (probabilité $\frac{1}{b_1}$), a_2 pour $\texttt{i} = b_2$, ... a_p si $\texttt{i} = b_p$ (probabilité $\frac{1}{b_p}$) et que `Random.int (i+1)` renvoie `i` (probabilité $\frac{1}{i}$) dans tous les autres cas.
   La probabilité d'obtenir la permutation σ sur `t` est donc le produit de ces probabilités (ce sont des évènements indépendants), c'est à dire $\frac{1}{n!}$.

Lorsque la totalité des éléments à rajouter n'est pas connue à l'avance, on peut utiliser une structure appelée **arbretas** qui est un arbre binaire (défini par `type 'a arb = V | N of 'a * 'a arb * 'a arb`) dont les noeuds sont étiquetés par des couples (élément, priorité), où la priorité est un nombre entier choisi uniformément au hasard au moment de l'ajout de l'élément. De plus :

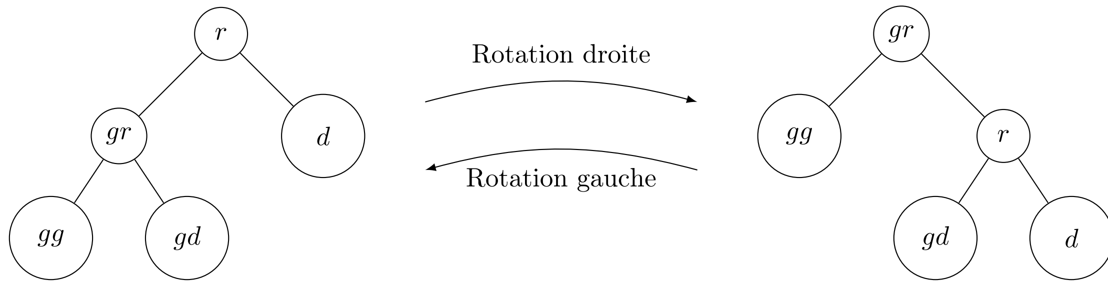
- les éléments doivent vérifier la propriété d'ABR.
- la priorité d'un sommet doit être inférieure à la priorité de ses éventuels fils (propriété de tas min sur les priorités).

3. Dessiner un arbretas dont les couples (élément, priorité) sont : (1, 4), (5, 6), (3, 8), (2, 2), (0, 7).



4. Étant donnés des éléments et priorités tous distincts, montrer qu'il existe un unique arbretas les contenants.
   ▶ Montrons P_n : « il existe un unique arbretas contenants n couples (élément, priorité) tous distincts » par récurrence sur n.
   P_0 est vraie : l'arbre vide convient et c'est le seul.
   Soit $n \in \mathbb{N}^*$ et supposons P_k vraie, $\forall k \leq n$. Considérons un ensemble C de $n+1$ couples (élément, priorité) tous distincts. Soit (e, p) le couple (élément, priorité) ayant la plus petite priorité, G les couples dont les éléments sont inférieurs à e et D les couples dont les éléments sont supérieurs à e. Les arbretas contenant les couples de C sont ceux s'écrivant $\text{N}((e, p), g, d)$ avec g et d arbretas contenant G et D. D'après l'hypothèse de récurrence, il existe un unique tel g et un unique tel d. Donc il existe aussi un unique arbretas contenant les couples de C.

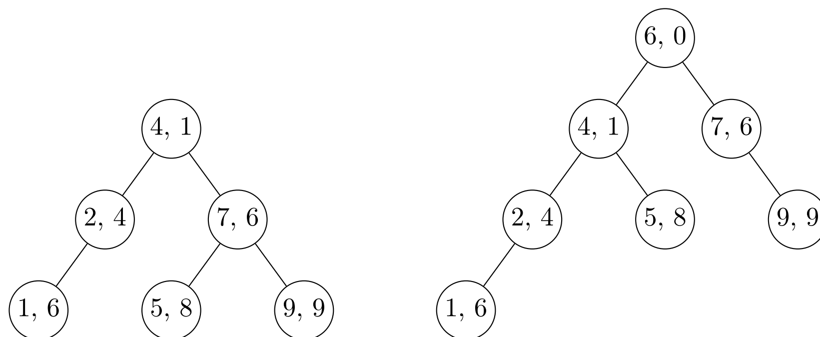
Nous allons utiliser des opérations de rotation sur un arbretas `N(r, N(gr, gg, gd), d)` :



5. Écrire une fonction `rotd` effectuant une rotation droite sur un arbre `N(r, N(gr, gg, gd), d)`.
   ▶ `let rotd = function N(r, N(gr, gg, gd), d) -> N(gr, gg, N(r, gd, d));;`

On supposera définie dans la suite une fonction `rotg` pour effectuer « l'inverse » d'une rotation droite. On remarquera que, si `a` est un ABR, `rotg a` et `rotd a` sont aussi des ABR.
Pour ajouter un sommet s dans un arbretas (en conservant la structure d'arbretas), on l'ajoute comme dans un ABR classique (en ignorant les priorités) puis, si sa priorité est inférieure à celle de son père, on applique une rotation sur son père pour faire remonter s et on continue jusqu'à rétablir la structure d'arbretas.

6. Dessiner l'arbretas obtenu en rajoutant (6, 0) à l'arbretas suivant :
   ▶ On obtient l'arbretas de droite.



7. Écrire une fonction utilitaire `prio` renvoyant la priorité de la racine d'un arbre (on renverra `max_int`, c'est à dire le plus grand entier représentable en base 2 sur le processeur, si cet arbre est vide).

```
let prio = function
  | V -> max_int
  | N(r, g, d) -> snd r;;
```

8. Écrire une fonction `add a e` ajoutant `e` (qui est un couple (élément, priorité)) à un arbretas `a`, en conservant la structure d'arbretas.
   ▶ `add` ajoute l'élément puis applique éventuellement une rotation sur le nouvel arbre :

```
let rec add a e = match a with
  | V -> N(e, V, V)
  | N(r, g, d) when e < r -> let g' = add g e in
                             if snd r < prio g' then N(r, g', d)
                             else rotd (N(r, g', d))
  | N(r, g, d) -> let d' = add d e in
                  if snd r < prio d' then N(r, g, d')
                  else rotg (N(r, g, d'));;
```

Pour supprimer un élément d'un arbretas, on commence par le chercher comme dans un ABR classique (en ignorant les priorités) puis on le fait descendre avec des rotations jusqu'à ce qu'il devienne une feuille qu'on peut alors supprimer librement.

9. Écrire une fonction `del` supprimant un élément dans un arbretas, en conservant la structure d'arbretas.
   ▶ Il faut effectuer la rotation sur le fils de priorité minimum pour conserver la structure. On utilise le fait que `prio V` renvoie `max_int` pour s'assurer qu'une rotation est appliquée sur un arbre convenable (par ex. `g` doit être non vide pour appliquer `rotd N(r, g, d)`). Noter l'élégance du 4ème cas ci-dessous :

```
let rec del a e = match a with
  | N(r, g, d) when e < fst r -> N(r, del g e, d)
  | N(r, g, d) when fst r < e -> N(r, g, del d e)
  | N(r, V, V) -> V (* e = r *)
  | N(r, g, d) -> del (if prio g < prio d then rotd a else rotg a) e;;
```

II Calcul de PPCM (Centrale 2007)

L'objectif de cette partie est d'analyser un algorithme permettant de calculer, pour $n \in \mathbb{N}$, le plus petit multiple commun de tous les entiers $\leqslant n$.
Il s'agit de $P_n = p_1^{\alpha_1} \dots p_k^{\alpha_k}$ avec $p_1, p_2, \dots, p_k$ la suite (strictement croissante) des nombres premiers compris au sens large entre 2 et n et pour chaque i, α_i est l'unique entier tel que $p_i^{\alpha_i} \leqslant n < p_i^{\alpha_i+1}$. Par exemple, $P_9 = 2^3.3^2.5.7$.
Plus précisément, on souhaite calculer tous les P_k, pour $k \in [\![1, n]\!]$, en exploitant au maximum les calculs précédents. Pour cela, on va utiliser une structure de tas.

Un tas est un arbre binaire dont les nœuds sont étiquetés par des éléments distincts d'un ensemble complètement ordonné. Chaque nœud possède une étiquette strictement plus *petite* que les étiquettes de ses éventuels fils. Les adjonctions et éventuelles suppressions de nœuds doivent préserver cette propriété, ainsi que le fait que *« tous les niveaux de l'arbre sont remplis, sauf éventuellement celui de profondeur h (hauteur de l'arbre), qui est lui même rempli de la gauche vers la droite »*. Par exemple, un tas possédant six nœuds sera constitué d'une racine, deux nœuds de profondeur 1, et 3 nœuds de profondeur 2. La structure de données utilisée en machine pour stocker et manipuler ces tas n'importe pas ici : on ne demande pas de programmer effectivement les algorithmes.

Ici, les nœuds sont étiquetés par des couples (p^α, p), avec p premier. Après le calcul de P_k, sont stockés dans le tas tous les couples d'entiers (p^α, p) tels que p est un nombre premier inférieur ou égal à k et α est le plus petit entier tel que $k < p^\alpha$. Par exemple, après avoir calculé P_9, on trouve dans le tas les couples $(16, 2)$, $(27, 3)$, $(25, 5)$ et $(49, 7)$.

Les couples sont ordonnés de la façon suivante : $(a, b) <_1 (a', b')$ si et seulement si $a < a'$. Pour cet algorithme, le tas ne contient jamais deux couples ayant la même première composante, de sorte que les étiquettes sont toujours comparables.

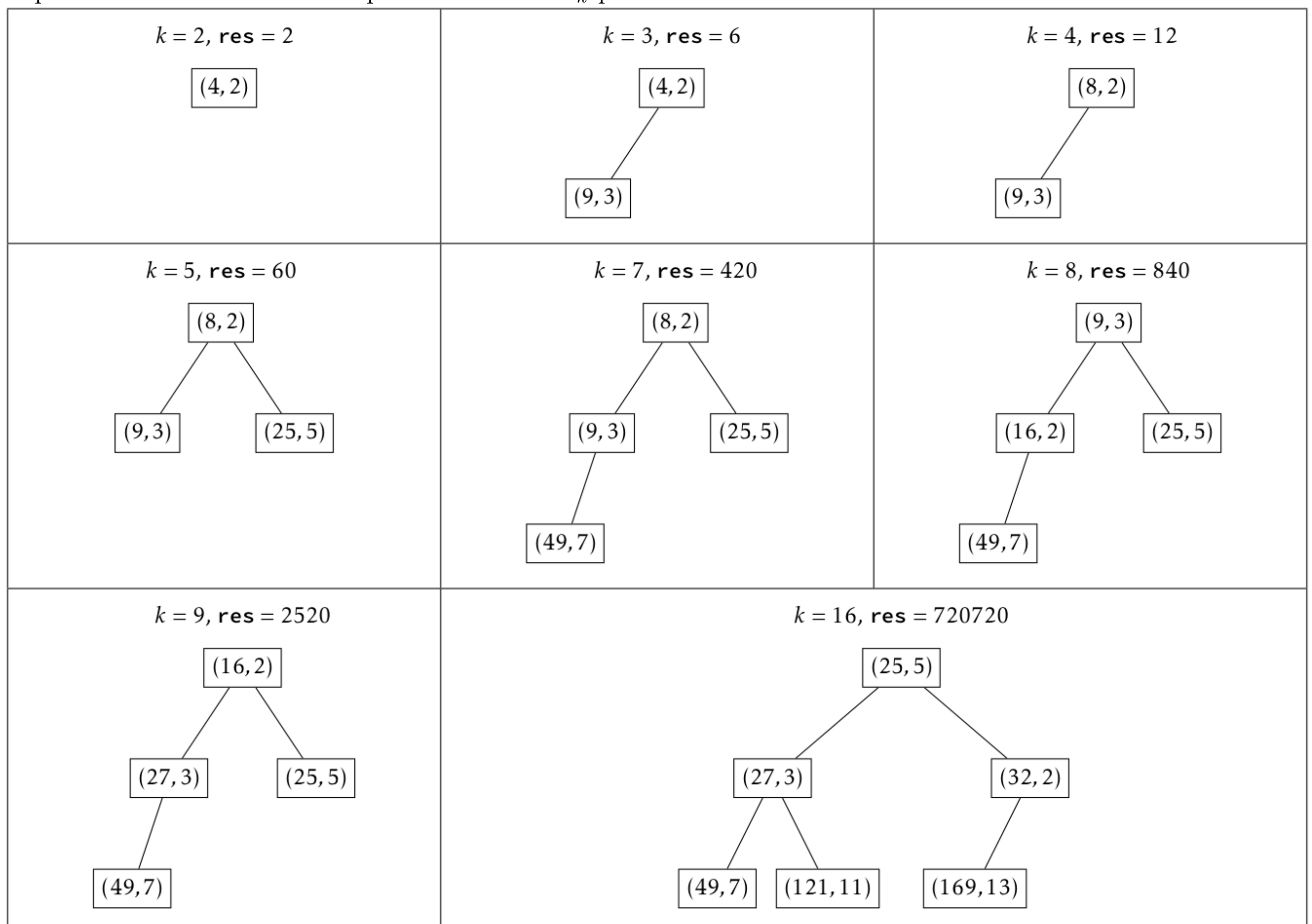
L'algorithme est itératif. On stocke dans une variable **Res** le ppcm calculé jusque là. Au départ **Res=2** est le ppcm des entiers $\leqslant 2$ et le tas est constitué d'un seul nœud indexé par $(4, 2)$. Après avoir calculé P_{k-1} (qui est alors présent dans **Res**), on calcule P_k de la façon suivante :

- Si k est premier, on multiplie **Res** par k, et on insère un nouveau nœud indexé par (k^2, k) dans le tas.
- Sinon, si la racine (p^α, p) est telle que $k = p^\alpha$, alors on multiplie **Res** par p, on change la racine en $(p^{\alpha+1}, p)$ et on reconstitue la structure de tas.

1. Comment insérer efficacement un nouveau sommet dans un tas (en préservant la structure de tas) ?
   ▶ Cours : l'ajouter en temps que dernière feuille, puis la permuter avec son père jusqu'à obtenir un tas.
2. Comment reconstituer efficacement la structure de tas après avoir changé la valeur de la racine ?
   ▶ Cours : on peut la permuter avec le plus petit de ses fils jusqu'à rétablir la condition de tas.

Par la suite, chacune des deux opérations décrites ci-dessus sera appelée une *percolation*.

3. Représenter les différentes tas après le calcul de P_k pour k allant de 3 à 9.



4. Montrer que la hauteur d'un tas à n sommets est $\lfloor \log_2(n) \rfloor$.
   ▶ cf cours.
5. Quel est le coût d'une percolation ? En admettant que $\ln(\mathrm{P}_n) \underset{\infty}{\sim} n$, montrer que le nombre de percolations effectuées pour le calcul de P_n est $\mathrm{O}(n)$.
   ▶ Le nombre d'échanges réalisés par une percolation est au plus la hauteur, i.e $\mathrm{O}(\log(n))$.
   Si $P_n = p_1^{\alpha_1} ... p_k^{\alpha_k}$, alors le nombre de percolations est $\displaystyle\sum_{i=1}^{k} \alpha_i \leq \sum_{i=1}^{k} \alpha_i \log_2(p_i) = \log(P_n) \sim n$, car $\log_2(p_i) \geq 1$ si $p_i \geq 2$.
6. Écrire une fonction `Caml` permettant de déterminer si un entier est premier. Quelle est sa complexité ? Existe-t-il une méthode plus efficace pour trouver tous les entiers premiers inférieurs à un entier n ?
   ▶ On peut chercher un diviseur de p entre 2 et $\sqrt{p}$, en $\mathrm{O}(\sqrt{p})$:

```
let premier p =
  let rec aux k =
    if k * k > p then true
    else p mod k <> 0 && aux (k+1) in
  aux 2;;
```

   Le crible d'Ératosthène serait plus efficace pour calculer tous les nombres premiers $\leq n$.
7. Déterminer la complexité en fonction de n pour calculer P_n en utilisant l'algorithme présenté dans ce problème.
   ▶ Bilan des complexités :
   - construire le tas : $\mathrm{O}(n\log(n))$
   - déterminer si les entiers $2 \leq k \leq n$ sont premiers : $\sum_{k=2}^{n} O(\sqrt{k}) = O(\sum_{k=2}^{n} \sqrt{k})$. Par comparaison série-intégrale, on trouve $\sum_{k=2}^{n} \sqrt{k} \sim \int_2^n \sqrt{t}dt \sim n\sqrt{n}$ et donc $\sum_{k=2}^{n} O(\sqrt{k}) = \mathrm{O}(n\sqrt{n})$ (On obtiendrait $\mathrm{O}(n \log\log(n))$ avec le crible d'Ératosthène, cf ci-dessous).
   - faire des percolations : $\mathrm{O}(n)$ par la question 5.

   La complexité est donc $\mathrm{O}(n\log(n)) + \mathrm{O}(n\sqrt{n}) + \mathrm{O}(n) = \mathrm{O}(n\sqrt{n})$ ($\mathrm{O}(n\log(n))$ avec le crible d'Ératosthène).
   $\Big($Preuve que le crible d'Ératosthène est $\mathrm{O}(n\log(\log(n)))$, en admettant le **théorème des nombres premiers**, affirmant que le $k$ème nombre premier p_k est équivalent à $k\ln(k)$:
   Le crible d'Ératosthène parcourt les entiers de 1 à n et, quand il voit un entier non rayé p, en déduit qu'il est premier et raye tous ses multiples (qui sont au nombre de $\frac{n}{p}$).
   Ainsi, le nombre total de « rayures » est $\displaystyle\sum_{p_k \text{ premier } \leq n} \frac{n}{p_k} = n \sum_{p_k \text{ premier } \leq n} \frac{1}{p_k}$.
   Comme $p_k \sim k\ln(k)$ et $x \longmapsto \frac{1}{x\ln(x)}$ est $\nearrow$, par comparaison série-intégrale :

$$\sum_{p_k \text{ premier } \leq n} \frac{1}{p_k} \sim \int^n \frac{1}{x\ln(x)} = \Big[\ln(\ln(x))\Big]^n \sim \ln\ln(n)$$

   D'où $\displaystyle\sum_{p_k \text{ premier } \leq n} \frac{n}{p_k} \sim n\ln(\ln(n)).\Big)$

III Arbre binaire de recherche optimal

Étant donnés des éléments $e_0 < e_1 < ... < e_{n-1}$ de probabilités d'apparitions p_0, ..., p_{n-1}, on veut construire un arbre binaire de recherche (ABR) a contenant e_0, ..., e_{n-1} et minimisant la complexité moyenne de recherche d'un élément (le *coût* $C(a)$ de a), qui est ($prof(e_i)$ étant la profondeur de e_i dans a) :

$$C(a) = \sum_{i=0}^{n-1} p_i(1 + prof(e_i))$$

1. Écrire une fonction `cout` ayant un ABR a en argument et renvoyant $C(a)$ en complexité linéaire en le nombre de sommets de a. On supposera définie, dans cette question, une fonction `proba` renvoyant en $\mathrm{O}(1)$ la probabilité d'apparition d'un élément de l'ABR.
   ▶ On ajoute la profondeur (+ 1) en argument. `cout 1` est la fonction demandée.

```
let rec cout prof a = match a with
     | V -> 0.
     | N(r, g, d) -> (proba r) *. prof +. (cout (prof +. 1.) g) +. (cout (prof +. 1) d);;
```

   Il y a un appel récursif (en temps constant) par élément de `a`, donc `cout 1` est linéaire.

Dans le reste de l'exercice, on veut calculer l'ABR optimal (de coût minimum) par programmation dynamique.
Pour cela on note :

- $w_{i,l} = p_i + ... + p_{i+l-1}$
- $c_{i,l}$ le coût de l'ABR optimal $a_{i,l}$ contenant e_i, ..., e_{i+l-1}

2. Montrer que $c_{i,l} = \min_{0 \leq k < l} (w_{i,l} + c_{i,k} + c_{i+k+1,l-k-1})$.
   Indice : que vaut $c_{i,l}$ si $a_{i,l}$ a pour racine e_{i+k} ?
   ▶ si $a_{i,l}$ a pour racine e_k alors son sous-arbre gauche g est un ABR optimal contenant e_i, ..., e_{k-1} : sinon, on pourrait le remplacer par un tel ABR optimal, ce qui contredirait l'optimalité de $a_{i,l}$. De même pour le sous-arbre droit, d'où

$$c_{i,l} = \underbrace{w_{i,k} + c_{i,k}}_{\text{coût de } g} + p_{i+k} + \underbrace{w_{i+k+1,l-k-1} + c_{i+k+1,l-k-1}}_{\text{coût de } d}$$

$$c_{i,l} = w_{i,l} + c_{i,k} + c_{i+k+1,l-k-1}$$

   La dernière égalité venant du fait que $w_{i,l} = w_{i,k} + p_{i+k} + w_{i+k+1,l-k-1}$.
   Dans le cas général, $c_{i,l}$ est la valeur minimum obtenue pour les différents k possibles :

$$\boxed{c_{i,l} = \min_{0 \leq k < l} (w_{i,l} + c_{i,k} + c_{i+k+1,l-k-1})}$$

3. Écrire une fonction `opt` ayant un tableau des probabilités d'apparitions en entrée et renvoyant le coût de l'ABR optimal correspondant. On pourra utiliser `Array.make_matrix n p 0.` pour créer une matrice $n \times p$ nulle.
   ▶ On remplit progressivement des matrices `w` et `c` (pour que `w.(i).(l)` contienne $w_{i,l}$ et que `c.(i).(l)` contienne $c_{i,l}$). On renvoie $c_{0,n}$.

```
let opt p =
  let n = Array.length p in
  let w = Array.make_matrix n (n + 1) 0. in
  let c = Array.make_matrix n (n + 1) 0. in
  for l = 1 to n do
    for i = 0 to n - l do
      w.(i).(l) <- w.(i).(l-1) +. p.(i + l - 1);
      c.(i).(l) <- w.(i).(l) +. c.(i).(l - 1);
      for k = 0 to l - 2 do (* calcul de minimum *)
        if w.(i).(l) +. c.(i).(k) +. c.(i + k).(l - k - 1) < c.(i).(l)
        then c.(i).(l) <- w.(i).(l) +. c.(i).(k) +. c.(i + k + 1).(l - k - 1)
      done
    done
  done;
  c.(0).(n);;
```

4. Quelle est la complexité de `opt` ?
   ▶ `Array.make_matrix n (n+1) 0.` demande $\Theta(n^2)$. Chaque `for` s'exécute au plus n fois, donc les 3 `for` imbriqués ont une complexité totale $\mathrm{O}(n^3)$, ce qui est aussi la complexité de `opt` ($\mathrm{O}(n^2) + \mathrm{O}(n^3) = \mathrm{O}(n^3)$).
5. Modifier `opt` de façon à renvoyer aussi l'ABR optimal.
   ▶ On peut stocker dans une matrice `a` les ABR optimaux `a.(i).(l)` et les remplir en même temps que `c.(i).(l)`.

IV Conversion d'arbre binaire de recherche en tas

On définit un arbre binaire par : `type 'a arb = V | N of 'a * 'a arb * 'a arb;;`
On représente un tas par un tableau comme dans le cours.

1. Écrire une fonction `infixe : 'a arb -> 'a list`, si possible en complexité linéaire, renvoyant la liste des sommets d'un arbre parcourus dans l'ordre infixe.
   ▶ On peut utiliser un accumulateur pour éviter `@`, comme dans le TD 1. On effectue un cons par élément de l'arbre, d'où une complexité linéaire.

```
let infixe a =
  let rec aux acc = function
  | V -> acc
  | N(r, g, d) -> aux (r::aux acc d) g in
  aux [] a;;
```

2. Écrire une fonction `tas_of_abr : 'a arb -> 'a array` telle que, si `a` est un arbre binaire de recherche à n éléments, `tas_of_abr a` renvoie, en $\mathrm{O}(n)$, un tableau représentant un tas min avec les mêmes éléments que `a`. On pourra utiliser `Array.of_list` qui convertit une `list` en `array`.
   ▶ Le tableau du parcours infixe d'un ABR est croissant, donc représente un tas min (on a bien `t.(i) < t.(2*i+1)` et `t.(i) < t.(2*i+2)`).
   `let tas_of_abr a = Array.of_list (infixe a);;`

3. Écrire une fonction `abr_of_tas : 'a array -> 'a arb` en $O(n \log(n))$ telle que, si `t` représente un tas min à n éléments (et $n =$ `Array.length t`), `abr_of_tas t` renvoie un arbre binaire de recherche presque complet et avec les mêmes éléments que `t`. On pourra utiliser directement une fonction `take: 'a array -> 'a` pour renvoyer et supprimer la racine d'un tas, comme dans le cours.
   ► on peut trier `t` (en n `take` soit $O(n \log(n))$, puis construire l'ABR en $O(n)$ par dichotomie (la racine est le milieu du tableau, le sous-arbre gauche la partie gauche du tableau et le sous-arbre droit la partie droite du tableau) pour obtenir un arbre presque complet (cf exercice de TD), donc de hauteur $O(\log(n))$:
```
let abr_of_tas t =
     let n = Array.length t in
     let tab = Array.make n t.(0) in (* tab va être un tri de t *)
     for i = 0 to n - 1 do tab.(i) <- take t done;
     let rec aux i j = (* convertit tab.(i), ..., tab.(j-1) en ABR  *)
          if i >= j then V
          else let m = (i + j) / 2 in
               N(tab.(m), aux i m, aux (m+1) j)
     in aux 0 n;;
```
   Pour vérifier qu'on obtient bien un arbre binaire presque complet (i.e. que tous les niveaux sont complets sauf éventuellement le dernier), on peut montrer par induction structurelle que, si a est un arbre binaire dont tous les noeuds $N(r, g, d)$ vérifient $|n(g) - n(d)| \leq 1$ (où $n(g)$ est le nombre de noeuds de g) alors a est presque complet.
4. Peut-on faire mieux que $O(n \log(n))$ pour la question précédente ?
   ► Si c'était le cas, on pourrait trier un tableau `t` de taille n en $o(n \log(n))$ (ce qui est impossible) :
   i) transformer `t` en tas min, avec l'algorithme linéaire du cours, en $O(n)$
   ii) convertir ce tas en ABR, en $o(n \log(n))$
   iii) renvoyer le parcours infixe de cet ABR, en $O(n)$