Tri par tas binaires

L'idée de cet algorithme de tri consiste à *structurer les données*, c'est-à-dire à les organisée dans une structure non triviale fondée sur les arbres, les tas binaires, et codée de façon très simple dans de banales listes.
L'algorithme obtenu est asymptotiquement optimal dans le cas le pire. En moyenne, l'algorithme se révèle environ deux fois moins efficace que le tri rapide, mais il ne présente pas le défaut d'avoir une complexité dans le cas le pire en $O(n^2)$, ce qui semble avoir été utilisé pour attaquer certains systèmes informatiques (les bloquer en lançant des listes dans lequel l'algorithme de tri rapide explose).

1 Arbres binaires presque complets

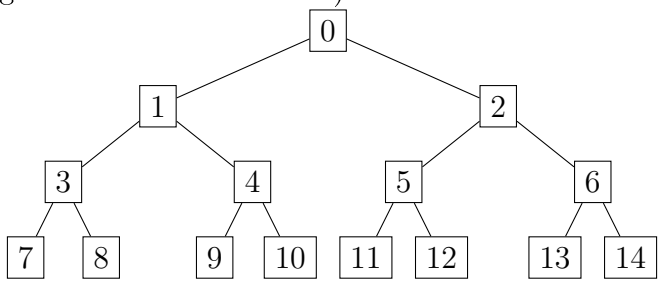
1.1 Vocabulaire des arbres

Un *graphe* est un ensemble (fini) de sommets reliés par des arêtes. Formellement, c'est un couple $\Gamma = (\Gamma_0, \Gamma_1)$ formé par un ensemble fini Γ_0 et une partie Γ_1 de paires de points de Γ_0. On appelle *sommets* ou *nœuds* de Γ les éléments de Γ_0 et *arêtes* les éléments de Γ_1. On dit qu'une arête $\alpha = \{i, j\}$ relie les nœuds i et j.
On appelle *chemin* toute suite finie de sommets reliés à leur successeur par une arête. En symboles, c'est une suite $(i_0, \dots, i_\ell)$ d'éléments de Γ_0 telle que pour un indice k entre 0 et $\ell-1$, la paire $\{i_k, i_{k+1}\}$ est une arête de Γ_1. On dit que le chemin relie i_0 et i_ℓ et que sa longueur est ℓ. Un chemin est un *cycle* si $i_0 = i_\ell$. Un chemin est *élémentaire* si tous les sommets qui le constituent sont tous distincts.
Un graphe est un *arbre libre* s'il est connexe et acyclique, c'est-à-dire si deux nœuds peuvent toujours être reliés par un chemin et s'il n'existe pas de cycle élémentaire de longueur non nulle. Dans un arbre libre, deux points peuvent toujours être reliés par un unique chemin élémentaire.
Un *arbre enraciné*, ou simplement *arbre*, est un arbre libre dans lequel on a choisi un nœud appelé *racine* : formellement, c'est un couple (Γ, r) où $\Gamma = (\Gamma_0, \Gamma_1)$ est un arbre libre et $r \in \Gamma_0$.
La *profondeur* d'un nœud j dans un arbre (Γ, r) est la longueur de l'unique chemin élémentaire qui relie r et j. Par exemple, la profondeur de la racine est 0, la profondeur de ses voisins est 1. La *hauteur* d'un arbre est la profondeur maximale d'un de ses nœuds. La *hauteur* d'un nœud est la différence entre la hauteur de l'arbre et la profondeur du nœud.
Le *père* d'un nœud j qui n'est pas la racine est l'avant-dernier nœud du chemin qui relie la racine à ce nœud : si $(r = i_0, \dots, i_{\ell-1}, i_\ell = j)$ est ce chemin, le père de j est $i_{\ell-1}$. On le notera à l'occasion $\pi(j)$ ou `pere(j)`. Un *fils* d'un nœud j est un nœud dont le père est j –si on veut, c'est un élément de $\pi^{-1}(j)$. Si un nœud j apparaît dans le chemin qui relie la racine à un nœud, on dit que ce nœud est un *descendant* de j. Par exemple, tout fils est un descendant et tout nœud est un descendant de la racine. Si un nœud n'a pas de fils, on dit que c'est une *feuille*, sinon on dit que c'est un *nœud interne*.
Un *arbre orienté* est un arbre dans lequel on choisit un ordre sur l'ensemble des fils de chaque nœud : formellement si un nœud j a f fils, on choisit une bijection $\{1, \dots, f\} \to \pi^{-1}(j)$.

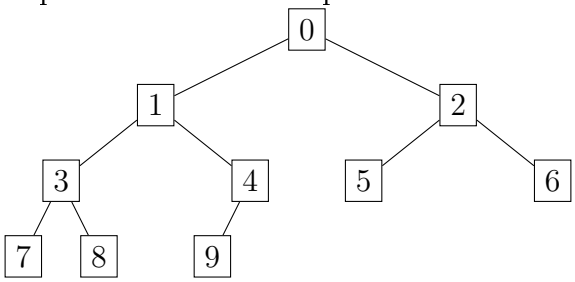
1.2 Arbres binaires (presque) complets

Dans ce qui suit, un *arbre binaire* est un arbre orienté où chaque nœud a au plus deux fils. Si un nœud a deux fils, celui qui correspond à l'indice 1 est appelé le *fils gauche*, celui qui correspond à

l'indice 2 le *fils droit*. S'il n'y a qu'un seul fils, c'est par défaut un fils gauche.
Un arbre binaire est *complet* si, h étant sa hauteur, tous les sommets de profondeur strictement plus petite que h ont exactement deux fils et tous les sommets de profondeur h sont des feuilles.
Il existe une façon standard de numéroter les nœuds d'un arbre binaire complet (parcours en largeur) : on commence par la racine, puis on décrit les sommets par profondeur croissante, de la gauche vers la droite à profondeur donnée (c'est-à-dire qu'on commence par les fils des premiers sommets de la couche précédente, le fils gauche avant le fils droit) :



Cela établit une façon d'indexer les nœuds de l'arbre par $\{0, 1, \ldots, 2^{h+1} - 1\}$.
Un arbre binaire *presque complet* à n nœuds est un arbre que l'on obtient en conservant les nœuds d'indices 0 à n d'un arbre complet ayant plus de n racines. Cela revient à dire que l'arbre est complètement rempli à tous les niveaux, sauf éventuellement le dernier qui est rempli en partant de la gauche et jusqu'à un certain point. Voici un exemple avec 10 sommets :



1. Vérifier que le nombre de nœuds d'un arbre binaire complet de hauteur h est $2^{h+1} - 1$.
2. Dans un arbre binaire complet de hauteur h, quels sont les indices des nœuds de profondeur k, $0 \leq k \leq h$? Quelle est la profondeur du nœud d'indice j, $0 \leq j < 2^{h+1} - 1$?
3. Quelle est la hauteur d'un arbre binaire presque complet à n nœuds ?
4. Si j est l'indice d'un nœud d'un arbre binaire presque complet, montrer que l'indice du fils gauche de j, du fils droit de j et du père de j sont, s'ils existent :

$$FG(j) = 2j + 1, \quad FD(j) = 2j + 2, \quad P(j) = \left\lfloor \frac{j-1}{2} \right\rfloor.$$

   Implémenter ces fonctions avec Xcas.
5. Dans un arbre binaire presque complet ayant n nœuds, montrer que les indices des feuilles sont $\lfloor (n+1)/2 \rfloor, \ldots, n-1, n$.
6. Dans un arbre binaire presque complet ayant n sommets, montrer que le nombre maximal de descendants d'un fils de la racine est $2n/3$. On pourra commencer par le cas où « la dernière ligne est remplie à moitié » (c'est-à-dire qu'il y a autant de feuilles de profondeur maximale et de profondeur plus petite), et montrer que c'est le cas le pire.

7. Dans un arbre binaire de hauteur h, il y a au plus 2^{h-k} nœuds de hauteur k.

Une résolution

1. On appelle « couche » un ensemble de nœuds ayant une profondeur donnée. La couche de la racine contient 1 nœud, la couche de profondeur $k+1$ contient les deux fils de chaque nœud de profondeur k. Par récurrence, il y a 2^k nœuds de profondeur k, soit

$$1 + 2^1 + \cdots + 2^h = 2^{h+1} - 1 \text{ nœuds en tout.}$$

2. D'après ce qui précède, il y a $2^k - 1$ nœuds de profondeur $< k$ et 2^k nœuds de profondeur k, ce qui correspond aux indices j satisfaisant à $(2^k - 1 + 2^k = 2^{k+1} - 1)$:

$$(*) \qquad 2^k - 1 \leq j < 2^{k+1} - 1.$$

   Étant donné un indice j, la profondeur du nœud j est l'unique entier k tel que l'inégalité $(*)$ soit satisfaite. Elle équivaut à :[1]

$$k \leq \log(j+1) < k+1, \quad \text{soit : } k = \lfloor \log(j+1) \rfloor.$$

3. La hauteur d'un arbre binaire presque complet à n nœuds est la profondeur du nœud indexé par $n-1$. C'est donc :

$$h = \lfloor \log n \rfloor.$$

4. Les formules proposées sont valables pour les premières valeurs de j. Soit j un nœud de profondeur $k = \lfloor \log(j+1) \rfloor$. Dans la couche de profondeur k, il y a $\ell = j - (2^k - 1)$ sommets d'indice inférieur à j. Leurs fils seront les sommets de la couche de profondeur $k+1$ qui seront numérotés avant les fils de j ; ils occuperont donc 2ℓ indices à partir de $2^{k+1} - 1$, si bien que le fils gauche de j sera indexé par :

$$FG(j) = 2^{k+1} - 1 + 2\ell = 2^{k+1} - 1 + 2\left(j - (2^k - 1)\right) = 2j + 1.$$

   D'évidence, le fils droit aura l'indice suivant : $FD(j) = FG(j) + 1$. La formule pour $P(j)$ s'en déduit en constatant qu'elle est valable pour les indices pairs et les indices impairs.

Xcas

```
f_gauche(j) := 2*j+1 ;
f_droit(j)  := 2*j+2 ;
pere(j)     := floor((j-1)/2) ;
```

5. On se convainc d'un regard que les feuilles correspondent aux « derniers indices ». L'indice minimal d'une feuille est le successeur du père de la feuille d'indice n, soit

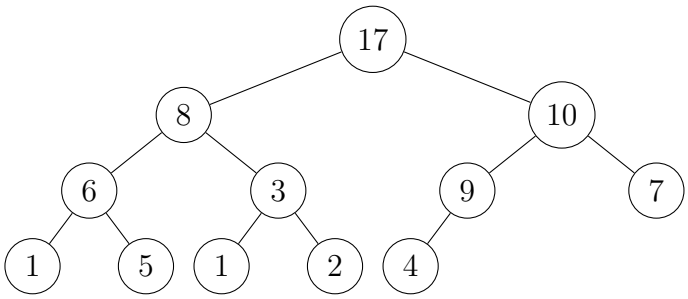
$$\left\lfloor \frac{n-1}{2} \right\rfloor + 1 = \left\lfloor \frac{n+1}{2} \right\rfloor.$$

[1] Ici, log est le logarithme à base 2 et $\lfloor \cdot \rfloor$ est la partie entière.

6. Fixons un entier h strictement positif et considérons les arbres binaires presque complets de profondeur h. On commence par celui dont la couche de profondeur h est remplie à moitié. Il y a donc autant de nœuds de profondeur h que de nœuds de profondeur $h-1$ d'où un nombre total de nœuds égal à :
$$n_0 = 2^h - 1 + 2^{h-1} = 3 \cdot 2^{h-1} - 1.$$
Le nombre de nœuds d'un sous-arbre est maximal pour le sous-arbre des descendants du fils gauche de la racine. Celui-ci est un arbre binaire complet de profondeur $h-1$, le nombre de nœuds qu'il contient est :
$$n' = 2^h - 1 = 2 \cdot 2^{h-1} - 1 = 2\frac{n_0+1}{3} - 1 \leq \frac{2n_0}{3}.$$
La propriété est donc prouvée dans ce cas. Si la couche de profondeur h est remplie moins qu'à moitié, disons qu'elle contient ℓ nœuds avec $\ell \leq 2^{h-1}$. Il y a donc $n = 2^h - 1 + \ell$ nœuds dans l'arbre et au plus $m2^{h-1} - 1 + \ell$ nœuds dans le sous-arbre (au pire, c'est le fils gauche de la racine). Par décroissance de l'homographie (de ℓ) qui en jeu, on a :
$$\frac{m}{n} = \frac{2^{h-1}+\ell}{2^h - 1 + \ell} \leq \frac{2^{h-1} - 1 + 2^{h-1}}{2^h - 1 + 2^{h-1}} = \frac{n'}{n_0} \leq \frac{2}{3}.$$
Si $\ell > 2^{h-1}$, c'est encore plus facile, car le nombre de nœuds du sous-arbre est au pire $m = 2^{h-1} - 1 + 2^{h-1}$ alors que le nombre de sommets de l'arbre est $n = 2^h - 1 + \ell > 2^h - 1 + 2^{h-1}$, ce qui permet de conclure.
Ces inégalités traduisent que le rapport du nombre de nœuds du sous-arbre à celui de l'arbre est bien atteint lorsque la dernière couche est remplie à moitié.
7. Évident par ce qui précède...

2 Tas binaires

Un *arbre binaire étiqueté* est la donnée d'un arbre binaire et d'une application définie sur l'ensemble des nœuds. Les images des nœuds seront appelés les étiquettes : ce sont des réels, des entiers, des chaînes de caractères... On supposera pouvoir comparer les étiquettes, c'est-à-dire disposer d'une relation d'ordre sur l'ensemble image.
Un *tas binaire* est un arbre binaire étiqueté auquel on associe un entier inférieur au nombre de sommets t qu'on appelle sa *taille*, et qui satisfait à la *propriété du tas* : l'étiquette d'un nœud j est supérieure à l'étiquette de ses fils dont l'indice n'est pas plus grand que la taille :
$$A[j] \geq A[FG(j)] \text{ et } A[j] \geq A[FD(j)].$$
Si un des fils de j est manquant ou s'il a un indice plus grand que la taille, on oublie la condition correspondante. En d'autres termes, pour la condition de tas, on ne considère que le sous-arbre des t nœuds d'indices les plus petits.
Voici un exemple de tas binaire dont la taille est le nombre de nœuds :



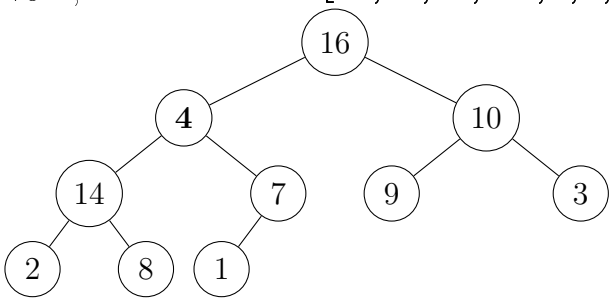
2.1 Tas binaire : entasser

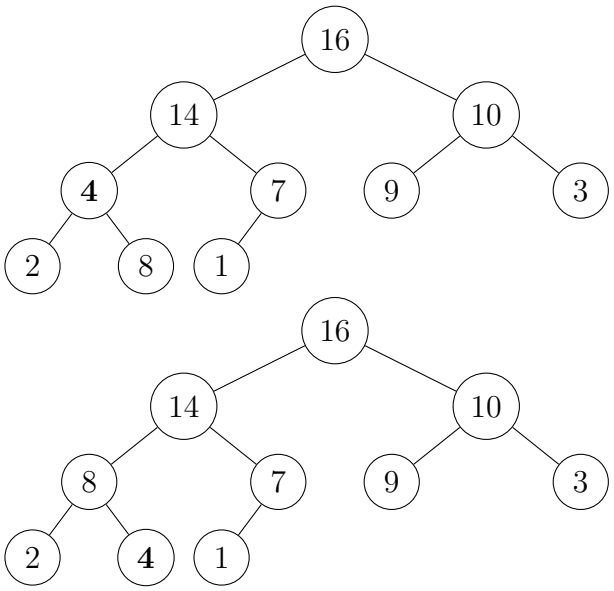
La première procédure utile est donnée ainsi :

Pseudo-code

```
def entasser_max(A,t,j):
    /* A un tableau, t sa taille, j le noeud où on entasse */
    /* calcul des fils du noeud j */
    l := gauche(j)
    r := droit(j)
    /* trouver l'indice m dont l'étiquette est maximale parmi j,r,l */
    si l <= t et A[r] > A[m]
       alors m := l
       sinon m := j
    si r <= t et A[r] > A[m]
       alors m := r
    /* l'indice d'étiquette maximale est m */
    /* on met l'étiquette maximale à la bonne place
       et on rétablit la propriété de tas */
    si m <> j
       alors echanger A[j] et A[m]
             entasser_max(A,t,m)
```

Voici, sur le tableau `A=[16, 4,10, 14,7,9,3, 2,8,1]`, l'effet de `entasser_max(A,11,1)`.





1. Examiner l'effet de la procédure `entasser_max(A,14,2)` sur le tableau
   `A=[27, 17,3, 16,13,10,1, 5,9,12,4,8,9,0]`.
   (Attention, le tableau est indexé de 0 à 13).
2. Pourquoi la procédure s'arrête-t-elle ?
3. On suppose que dans l'arbre d'entrée, les arbres des descendants des fils r et ℓ de j ont la propriété du tas. Montrer qu'en sortie de fonction, l'arbre des descendants de j a la propriété du tas.
4. On note c_n le nombre maximal de comparaisons effectuées pendant le calcul de la fonction `entasser_max` à partir d'un arbre à n nœuds. Démontrer que l'on a pour tout n :
   $$c_n \leq 2 + c_{2n/3},$$
   où $c_{2n/3}$ est un raccourci pour $c_{\lfloor 2n/3 \rfloor}$.
5. Montrer qu'il existe deux constantes positives K et L telles que pour tout entier n, on ait :
   $$c_n \leq K \log n + L.$$
6. Traduire le pseudo-code en Xcas.

Une résolution

1. Hem...
2. Dans la procédure récursive `entasser_max`, la hauteur du nœud j auquel on l'applique diminue à chaque appel.
3. Tout d'abord, remarquons que la procédure ne fait que permuter des étiquettes sans changer la structure de l'arbre.

   On procède par récurrence sur la hauteur du sommet j. Soit h un entier, supposons que la propriété vraie pour h. Soit j un nœud de hauteur $h+1$. La procédure détecte l'étiquette maximale parmi celles du nœud j et de son ou ses fils, et la place à la racine du sous-arbre

des descendants de j (formé de j et son ou ses fils). Au niveau de la racine, la propriété de tas est satisfaite. Le sous-arbre des descendants du fils de j qui a hérité de l'étiquette de j se voit appliquer la procédure : par hyptohèse de récurrence, en sortie de procédure, il a la propriété du tas. L'autre sous-arbre éventuel l'a aussi car il n'a pas changé. On peut conclure la récurrence.

4. Dans la première partie de la procédure, on effectue 2 comparaisons. Puis on invoque éventuellement la procédure avec le sous-arbre des descendants d'un des fils de j. D'après la question 1.2 6 ci-dessus, cet arbre a au plus $2n/3$ sommets, donc on effectue au plus $c_{2n/3}$ comparaisons dans l'appel récursif.
5. On cherche une condition suffisante sur K et L pour que quel que soit n, on ait : $c_n \leq K \log n + L$. Comme pour tout n, on a : $\lfloor 2n/3 \rfloor < n$, il est suffisant que :

$$2 + K \log \frac{2n}{3} + L \leq K \log n + L, \quad \text{ou encore :} \quad K \geq 2/\log(3/2).$$

Supposons en effet que L soit choisi de sorte que $c_1 \leq L$ et que $K \geq 2/\log(3/2)$. Montrons par récurrence l'inégalité souhaitée. Pour $n = 1$, c'est clair. Soit $n \in \mathbb{N}$, on suppose que pour tout $k < n$, on a : $c_k \leq K \log k + L$. On a, d'après ce qui précède, vu que $k = \lfloor 2n/3 \rfloor < n$:

$$c_n \leq 2 + c_{2n/3} \leq 2 + K \log \frac{2n}{3} + L \leq K \log n + L,$$

et on conclut par récurrence.

Remarque : la même analyse fonctionne si on raisonne sur le temps au lieu du nombre de comparaisons, en remplaçant 2 par une constante (temps pour faire deux comparaisons et trois affectations). On peut retenir que cette procédure a une complexité en $O(\log n)$.

6. Voici une version en Xcas

Xcas

```
entasser_max(A,t,j) := {
  local l,r,m,c ;
  l := f_gauche(j) ;
  r := f_droit(j) ;
  m := j ;
  si l<t alors {
    si A[l]>A[j] alors { m := l ; } ;
  } ;
  si r<t alors {
    si A[r]>A[m] alors { m := r ; } ;
  } ;
  si m <> j alors {
    c := A[j] ;
    A[j] := A[m] ;
    A[m] := c ;
    A := entasser_max(A,t,m) ;
  } ;
  retourne A ;
} ;
```

Xcas

```
/* Test */
L := [16, 4,10, 14,7,9,3, 2,8,1] ;
entasser_max(L,size(L),1) ;
```

2.2 Construction de tas binaires

Partant d'un tableau quelconque, représentant un arbre binaire presque complet étiqueté, on veut le transformer en un tableau représentant un tas binaire en permutant les étiquettes.
On a vu que dans un arbre binaire presque complet à n nœuds, les feuilles portent les indices

$$\left\lfloor \frac{n+1}{2} \right\rfloor, \ldots, n-1, n.$$

On introduit la procédure suivante :

Pseudo-code

```
def construire_tas(A):
    n := size(A)
    pour j = floor((n-1)/2) jusque 1 pas -1
        faire entasser_max(A,n,j)
```

On va commencer par montrer que l'arbre binaire étiqueté obtenu en sortie de boucle a la propriété de tas, puis évaluer la complexité.

1. Montrer que l'assertion suivante est un invariant de boucle : le sous-tableau `A[j..n]`, correspondant aux indices j à n, possède la propriété de tas.
2. Montrer facilement que la complexité de cette procédure est en $O(n \log n)$.
3. Montrer (plus délicat) que la complexité est en $O(n)$. (L'idée est qu'il y a de nombreux sous-arbres de faible hauteur.)
4. Traduire le pseudo-code en Xcas.

Une résolution

1. À l'entrée dans la fonction, comme les nœuds portant les indices

$$\left\lfloor \frac{n+1}{2} \right\rfloor, \ldots, n-1, n$$

sont des feuilles, la propriété de tas est trivialement satisfaite. Cela permet d'amorcer une récurrence descendante sur j, en commençant par $j = \lfloor (n+1)/2 \rfloor$.
Pour l'hérédité, il suffit de remarquer que tous les descendants du nœud j sont dans le sous-tableau où la propriété de tas est satisfaite. En effet, on sait que si on invoque `entasser_max` sur un nœud dont les arbres descendants ont la propriété de tas, l'arbre étiqueté obtenu en sorti a aussi cette propriété.

2. On a vu que la complexité de la fonction `entasser_max` était majorée par une constante multipliée par le logarithme du nombre de nœuds (en fait, de la taille). On effectue environ $n/2$ appels à cette fonction, ce qui donne une complexité en $O(n \log n)$.
3. Considérons un arbre binaire presque complet à n sommet et soit $h = \lfloor n/2 \rfloor$ sa hauteur. On groupe les nœuds selon leur hauteur k, un entier compris entre 0 et h. Il y a au plus 2^{h-k} nœuds de hauteur k. Le sous-arbre des descendants d'un tel nœud au plus 2^k nœuds, donc la procédure `entasser_max` appliquée à un tel nœud a une complexité en $O(\log 2^k) = O(k)$. Au total, cela fait donc une complexité en

$$\sum_{k=0}^{h} 2^{h-k} O(k) = O\left(2^h \sum_{k=0}^{h} \frac{k}{2^k}\right) = O(n),$$

en utilisant le fait que $2^h = O(n)$ et que $\sum_{k=0}^{+\infty} k 2^{-k} = 2$.

4. Voici une façon de faire.

Xcas

```
construire_tas_max(A) := construire(A,size(A)) ;

construire(A,l) := {
  local ll,j ;
  ll := floor(l/2) ;
  pour j de ll+1 jusque 1 pas -1 faire
    A := entasser_max(A,size(A),j-1) ;
    afficher(A,j) ;
  fpour ;
  retourne A ;
} ;
```

Exemple

Xcas

```
L := [seq(irem(j*7,10),j=1..10)] ;
construire_tax_max(L) ;
```

2.3 Tri d'un tas

Voici enfin l'algorithme de tri par tas :

Pseudo-code

```
def trie_par_tas(A,t):
    construire_tas(A)
    pour t := size(A) jusque 2 pas -1
        faire echange A[0] et A[t-1]
              entasse_tas(A,t,0)
```

1. Montrer que l'assertion suivante est un invariant de la boucle `pour` : le sous-tableau `A[0..t-1]` correspondant aux indices de 0 à t a la propriété de tas et $A[0] \leq A[t] \leq A[t+1] \leq \cdots \leq A[n]$.
2. Montrer que la complexité de cet algorithme est, dans le cas le pire, $O(n \log n)$.
3. Traduire le pseudo-code en Xcas.

Une résolution

1. À l'entrée de la boucle, la première partie de l'assertion est vraie parce que le tableau est un tas et la deuxième partie en résulte (dans un tas, la racine porte l'étiquette maximale). Supposons que l'assertion soit vraie à l'entrée de la boucle pour une certaine valeur de t. À ce moment, on a $A[0] \leq A[t+1]$. De plus, comme `A[0..t-1]` est un tas, $A[0] \geq A[t]$. Par suite, après permutation, on a : $A[0] \leq A[t] \leq A[t+1] \leq \cdots \leq A[n]$, ce qui est la deuxième partie de l'assertion.

   On applique `entasser_tas`, ce qui ne modifie que le sous-tableau `A[0..t-1]` correspondant aux t premiers indices. Les arbres descendant des fils de la racine sont des tas : sous cette hypothèse, `entasser_max` rétablit la propriété de tas, ce qui prouve la première partie de l'assertion.
2. La construction du tas se fait avec une complexité $O(n)$. On itère moins de n fois la boucle, laquelle possède une complexité en $O(1)$ pour la permutation des étiquettes et $O(\log n)$ pour la procédure `entasser_max`. Au total, on a une complexité (dans le cas le pire) en $O(n \log n)$.
3. Voici une façon de faire.

Xcas

```
tri_tas(A) := tri(A,size(A)) ;
tri(A,l) := {
  local c,j ;
  pour j de l jusque 2 pas -1 faire
    c := A[0] ;
    A[0] := A[j-1] ;
    A[j-1] := c ;
    A := entasser_max(A,j-1,0) ;
  fpour ;
} ;
```